50.039 Theory and Practice of Deep Learning

W5-S2 Times Series Data and Recurrent Neural Networks

Matthieu De Mari

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

About this week (Week 5)

1. What is a **time series dataset**?
2. How to **analyze a time series**, and what are **typical deep learning models** capable of doing that?
3. What is **history**, and why is it needed for time series predictions?
4. What is a **good history length**?
5. What is **memory** and how to represent it in a Neural Network model?
6. How to implement a first **vanilla Recurrent Neural Network** model?
7. Why is the **vanishing gradient problem** prominent in RNNs?

About this week (Week 5)

8. What is the **LSTM** model?
9. What is the **GRU** model?
10. What are **one-to-one**, **one-to-many**, **many-to-one** and **many-to-many** models?
11. What are some **typical application examples** of these?
12. What is a **Seq2Seq** model?
13. What are **encoder** and **decoder** architectures?
14. What is an **autoregressive RNN**?

Defining a Recurrent Neural Network (RNN)

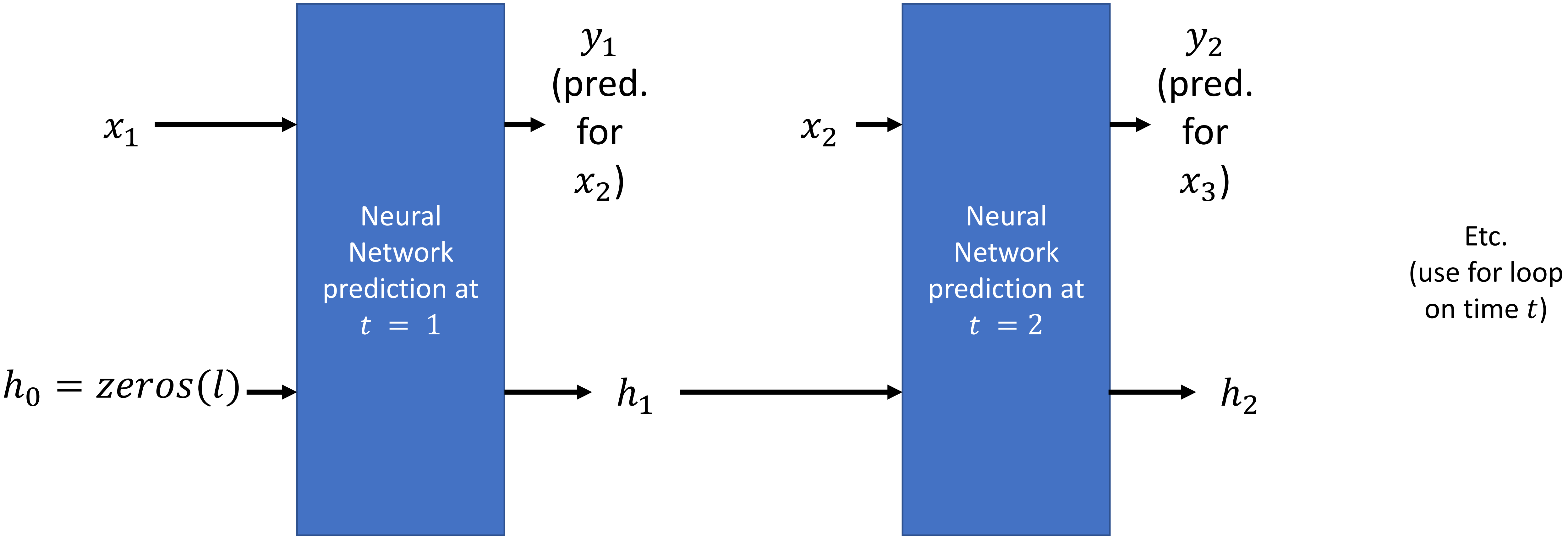
Definition (Recurrent Neural Network):

A **Recurrent Neural Network (or RNN)** is a neural network that operates on series and will:

- Receive an input which consists of **the observation x_t at time t**, and a **memory vector h_t computed as one of the Neural Network outputs at time $t - 1$**.
- Compute a **prediction y_t for what should match the value of x_{t+1}**, and an **updated memory vector h_{t+1}**, hopefully keeping a memory of what has happened in the previous operations.

The RNN is then used on all datapoints in the time series, using a for loop repeating the forward pass operation on all data points.

The need for memory



Implementing a RNN

Let us rewrite our first RNN model, which resembles the previous DNN with some changes:

- 3 Linear Layers + ReLU,
- No ReLU on final layer,
- **Number of inputs = 2, being (x_t, h_t) this time,**
- **Number of outputs = 2, being (y_t, h_{t+1}),**
- Hidden layers sizes: 32 and 8.

```
class RNN(torch.nn.Module):

    def __init__(self):
        super(RNN, self).__init__()
        self.layers = torch.nn.Sequential(torch.nn.Linear(2, 32),
                                          torch.nn.ReLU(),
                                          torch.nn.Linear(32, 8),
                                          torch.nn.ReLU(),
                                          torch.nn.Linear(8, 2))

    def forward(self, inputs, hidden):
        combined = torch.tensor([inputs, hidden]).to(inputs.device)
        out = self.layers(combined)
        return out
```

Notice how the forward method expects two values, being x_t (inputs) and h_t (hidden). They will be combined before going through the NN layers.

Implementing a RNN

Our trainer function is almost the same as before, except that:

- We initialize a hidden tensor with zero value,
- Outputs are split into y_t (in variable out) and the new hidden vector h_{t+1} (in variable hidden),
- Loss function uses y_t (in variable out) and target.

```python
def train(model, dataloader, num_epochs, learning_rate, device):
    criterion = torch.nn.MSELoss()
    optimizer = torch.optim.Adam(model.parameters(), lr = learning_rate)
    hidden = torch.tensor([0]).to(device)
    for epoch in range(num_epochs):
        loss = 0
        for inputs, targets in dataloader:
            optimizer.zero_grad()
            outputs = model(inputs.to(device), hidden)
            out, hidden = outputs[0], outputs[1]
            loss += criterion(out.to(device), targets.to(device))
        loss /= len(dataloader)
        loss.backward()
        optimizer.step()

        print(f"Epoch {epoch + 1}/{num_epochs}, Loss: {loss.item():.4f}")
```

Backpropagation through time

At the moment, we have a RNN, which, at each time t, will:

- Process each input x_t,
- Attempt to predict the value of x_{t+1},
- Keeps track of some memory, in a vector h_t, updated at each time t,
- Adjusts the parameters of the RNN layers at each time t.

This means that we perform **1 parameter update per sample** (as we would if we used **stochastic gradient descent** on a standard DNN).

We have however seen that using batches of **N data samples per parameter update** was better (a.k.a. as mini-batch gradient descent).

Backpropagation through time

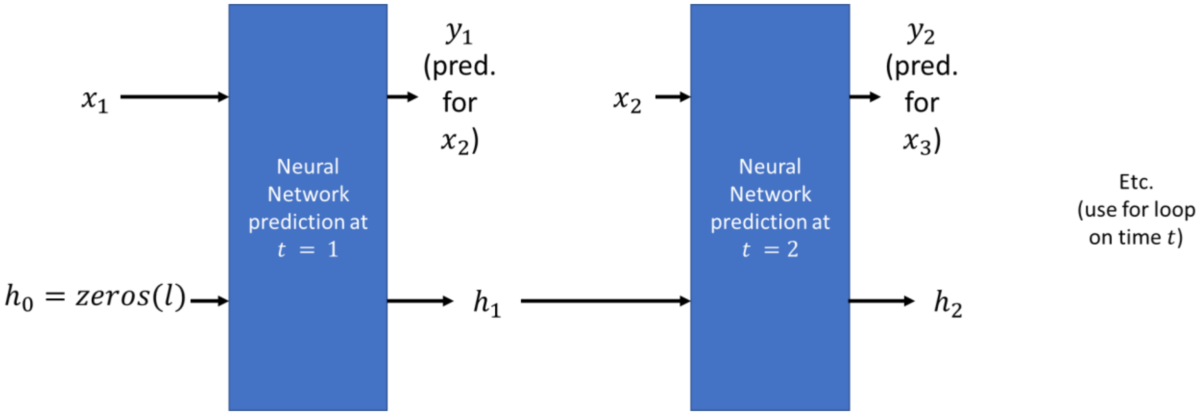
Definition (backpropagation through time):

In order to implement **a parameter update every N samples**, we need to implement a **backpropagation through time**.

- Unfold the RNN operations over time as shown,
- Run predictions on N consecutive samples and keep track of loss,
- Eventually perform one parameter update after N samples.

This basically allows to **teach the RNN how to update the memory vectors over time**.

Somewhat equivalent to a chain of successive layers, with gradient propagated through all the layers and time steps.



Backpropagation through time and the vanishing gradient problem

How to address the vanishing gradient problem in RNNs?

- Several techniques have been developed to address the vanishing gradient problem in RNNs, such as using alternative activation functions, initializing the weights of the network carefully, etc.
- **The “best” (?) solution requires using specialized architectures like Long Short-Term Memory (LSTM) networks or Gated Recurrent Units (GRUs) to process memory.**
- These are designed to better handle long-term dependencies, and can help to mitigate the vanishing gradient problem.
- This eventually improves the performance of RNNs on time series.

Core idea behind GRU and LSTM models

Our current RNN: uses the same layers and operations to both

- **Predict the x_{t+1} value,**
- **And update the memory vector, defining h_{t+1}.**

However, these are two very different operations and should probably rely on two distinct calculations!

```
class RNN(torch.nn.Module):

    def __init__(self):
        super(RNN, self).__init__()
        self.layers = torch.nn.Sequential(torch.nn.Linear(2, 32),
                                          torch.nn.ReLU(),
                                          torch.nn.Linear(32, 8),
                                          torch.nn.ReLU(),
                                          torch.nn.Linear(8, 2))

    def forward(self, inputs, hidden):
        combined = torch.tensor([inputs, hidden]).to(inputs.device)
        out = self.layers(combined)
        return out
```

Core idea behind GRU and LSTM models

Task 1: Predict the x_{t+1} value.

- This is the job of a "predictor".
- Uses information from memory h_t and the present observation x_t to formulate prediction.
- **Somewhat similar to predicting price of apartment based on apartment features.**
- For this, it is fine to use all the layers and operations we have implemented in previous weeks.

Core idea behind GRU and LSTM models

Task 2: update the memory vector, defining h_{t+1}.

- This typically consists of teaching the Neural Network how to remember relevant things that happened over time.
- **In other words, the cognitive ability of <u>memorizing and processing information over time</u>.**
- New task, probably requires its own set of operations!

Core idea behind GRU and LSTM models

But, hold on a second, what is the brain doing when it comes to memory anyway?

- Our brains will decide what is an important information to **remember** over time,
- But it might also choose to **forget** about information that is no longer relevant!
- (Basically, freeing space in the memory of your brain!)

Learn more about memory and the brain, here: https://lesley.edu/article/stages-of-memory

Restricted

Core idea behind GRU and LSTM models

The brain also has **several types of memories**. For instance, it has:

- A **short-term** memory (What did you have for lunch today? How about lunch two months ago?)
- A **long-term** memory (What was your favourite toy as a kid? What was the first video game you played?)

The brain seems capable to decide what makes it into each of these memories, in a **decorrelated** way.

Learn more about memory and the brain, here: https://lesley.edu/article/stages-of-memory

Core idea behind GRU and LSTM models

Some "neuroscience" ideas to improve our current vanilla RNN:

1. **Decorrelate both tasks!**
    - Have **distinct sets of trainable parameters** for the neural network (one for each of the two tasks)
    - And **different sets of operations happening in the forward method** (one for each task). And add operations that will nicely recombine these.
2. **Implement different types of memory (short and long).**
    - Requires new operations to be shown later.

Introducing LSTM

Definition (Long Short-Term Memory models):

LSTMs (Long Short-Term Memory models) are a type of recurrent neural network architecture introduced in [Hochreiter1997] and revised in [Gers2013].

LSTMs aim to solve the vanishing gradient problem in RNNs, by **adding gating mechanisms to selectively update and discard information in the cell state**. In other words, LSTMs use gating mechanisms to regulate the flow of information within the network.

This allows the model to **selectively choose which information from the previous cell state to retain and which to discard**, in order to give more importance to recently predicted values, for instance.

Introducing LSTM

The **cell state can be seen as a conveyor belt that carries information, or memory, across the network.**

- **The gates can either allow or prevent information from flowing through the belt, <u>mimicking this idea or remembering or forgetting things in the brain</u>.**
- This feature makes LSTMs better suited for modelling complex time series data that exhibit long-term dependencies, as they can selectively choose which information to remember or forget at each time step, depending on the input and the context.
- **Therefore, mimicking the behavior of the human brain!**

Introducing LSTM

Below are the equations used in a LSTM model.

- **Forget Gate:** will be used to decide what information to throw away from the previous cell state, c_{t-1}.

$$f_t = \sigma(W_f\, x_t + U_f\, h_{t-1} + b_f)$$

The presence of a sigmoid, forces f_t to have a value in $[0, 1]$.
(Will be important later on).

Similarly,

- **Input Gate:** will be used to decide what new information to store in the current cell state, c_t.

$$i_t = \sigma(W_i\, x_t + U_i\, h_{t-1} + b_i)$$

The presence of a sigmoid, forces f_t to have a value in $[0, 1]$.
(Will be important later on).

Introducing LSTM

Below are the equations used in a LSTM model.

- **Output Gate:** will be used to decide what information to output from the current cell state, c_t.

$$o_t = \sigma(W_o \, x_t + U_o \, h_{t-1} + b_o)$$

The presence of a sigmoid, forces f_t to have a value in $[0, 1]$.
(Will be important later on).

Important to note:

- All three equations for the forget, input and output gates seem to be following the same logic.
- However, they have their own sets of parameters (weights and biases), which can be trained independently later on.

Introducing LSTM

Below are the equations used in the forward method of a LSTM model.

- **New cell state:** The new cell state c_t, is then computed by:

$$c_t = f_t \, c_{t-1} + i_t \tanh(W_c \, x_t + U_c \, h_{t-1} + b_c)$$

The new cell state will consist of

- **A proportion $f_t \in [0, 1]$ to keep from the previous cell state c_{t-1}** (= "remembering" part of the previous memory state in the new c_t),
- **A new memory state**, decided as a function of the previous hidden state h_{t-1} and the current observation x_t. A certain proportion $i_t \in [0, 1]$, is then added to the new memory vector c_t.

Introducing LSTM

Below are the equations used in the forward method of a LSTM model.

- **New cell state:** The new cell state c_t, is then computed by:

$$c_t = f_t \, c_{t-1} + i_t \tanh(W_c \, x_t + U_c \, h_{t-1} + b_c)$$

Question: why use **tanh()** here instead of sigmoid?

- We do not want the values in c_t to be positive real values only. But at the same time, we do not want these values to go crazy (normalize!).
- Using tanh() keeps values in [-1, 1] and the gates keep the values in c_t under control and somewhat normalized.

Introducing LSTM

Below are the equations used in the forward method of a LSTM model.

- **New hidden state:** The new hidden state h_t, is computed by:

$$h_t \; = \; o_t \; tanh(c_t)$$

The new cell state will therefore simply consist of a proportion $o_t \in [0,1]$ of the current cell state c_t.

Important question: Hold on a second, why do we need two parameters, c_t and h_t to keep track of the memory?!

Introducing LSTM

Important question: But wait, why do we need two parameters, c_t and h_t to keep track of the memory?!

- They serve different purposes in the model…!
- The **cell state** acts as a first type of memory for the LSTM.
- In fact, it is **responsible for retaining long-term information** over time. It is the "state" of the cell and is passed from one time step to the next, allowing the LSTM to **maintain a longer-term memory**.

Introducing LSTM

Important question: But wait, why do we need two parameters, c_t and h_t to keep track of the memory?!

- The **hidden state**, on the other hand, is used to **output the prediction** and **capture shorter-term dependencies in the data**.
- It serves as a second type of memory that allows the model to **capture patterns in the input data over shorter time periods**.
- Separating both helps the model avoid the vanishing gradient problem and, at the same time, allows it to better capture short- and long-term dependencies in the data.

Introducing LSTM

Below are the equations used in the forward method of a LSTM model.

- **Prediction output:** The predicted output y_t, supposed to match the ground truth x_{t+1} is simply computed by reusing h_t:

$$y_t = h_t$$

- More often than not, however, we prefer to compute the prediction y_t as a Linear operation using h_t as input, that is:

$$y_t = Linear(h_t)$$

To recap, our LSTM

x_t → LSTM → y_t (pred. for x_{t+1})

h_{t-1}, c_{t-1} → LSTM → h_t, c_t

LSTM Neural Network prediction at time t

$$f_t = \sigma(W_f\, x_t + U_f\, h_{t-1} + b_f)$$
$$i_t = \sigma(W_i\, x_t + U_i\, h_{t-1} + b_i)$$
$$o_t = \sigma(W_o\, x_t + U_o\, h_{t-1} + b_o)$$
$$c_t = f_t\, c_{t-1} + i_t \tanh(W_c\, x_t + U_c\, h_{t-1} + b_c)$$
$$h_t = o_t\, tanh(c_t)$$
$$y_t = Linear(h_t) \text{ or } y_t = h_t$$

Building our own LSTM model

Let us start by defining all the weights and biases we need for each of the 6 operations we discussed.

We implement them in the init method of our class as before.

Pay attention to the sizes used for each parameter.

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

        # Parameters for the forget gate
        self.Wf = torch.nn.Parameter(torch.randn(input_size, hidden_size))
        self.Uf = torch.nn.Parameter(torch.randn(hidden_size, hidden_size))
        self.bf = torch.nn.Parameter(torch.zeros(hidden_size))
        # Parameters for the input gate
        self.Wi = torch.nn.Parameter(torch.randn(input_size, hidden_size))
        self.Ui = torch.nn.Parameter(torch.randn(hidden_size, hidden_size))
        self.bi = torch.nn.Parameter(torch.zeros(hidden_size))
        # Parameters for the cell gate
        self.Wc = torch.nn.Parameter(torch.randn(input_size, hidden_size))
        self.Uc = torch.nn.Parameter(torch.randn(hidden_size, hidden_size))
        self.bc = torch.nn.Parameter(torch.zeros(hidden_size))
        # Parameters for the output gate
        self.Wo = torch.nn.Parameter(torch.randn(input_size, hidden_size))
        self.Uo = torch.nn.Parameter(torch.randn(hidden_size, hidden_size))
        self.bo = torch.nn.Parameter(torch.zeros(hidden_size))

        # Parameters for the prediction
        self.V = torch.nn.Parameter(torch.randn(hidden_size, output_size))
        self.b = torch.nn.Parameter(torch.randn(1, output_size))
```

Building our own LSTM

Eventually, compute all 6 LSTM operations in the forward method.

```
def forward(self, inputs, cell_state, hidden_state):

    # Compute the forget gate
    forget_gate = torch.sigmoid(torch.matmul(inputs, self.Wf) + torch.matmul(hidden_state, self.Uf) + self.bf)
    # Compute the input gate
    input_gate = torch.sigmoid(torch.matmul(inputs, self.Wi) + torch.matmul(hidden_state, self.Ui) + self.bi)
    # Compute the output gate
    output_gate = torch.sigmoid(torch.matmul(inputs, self.Wo) + torch.matmul(hidden_state, self.Uo) + self.bo)

    # Compute the cell gate
    candidate_cell = torch.tanh(torch.matmul(inputs, self.Wc) + torch.matmul(hidden_state, self.Uc) + self.bc)
    # Compute the updated cell state
    cell_state = forget_gate * cell_state + input_gate * candidate_cell

    # Compute the updated hidden state
    hidden_state = output_gate * torch.tanh(cell_state)

    # Compute the output
    output = torch.matmul(hidden_state, self.V) + self.b

    return cell_state, hidden_state, output
```

Restricted

Building our own LSTM

We can then quickly check it has the expected behaviour, with consistency on the sizes.

```
# Testing out LSTM model
lstm = LSTM(input_size = 1, hidden_size = 4, output_size = 1)
input_data = torch.from_numpy(np.random.randn(1, 1)).float()
h_data = torch.from_numpy(np.random.randn(1, 4)).float()
c_data = torch.from_numpy(np.random.randn(1, 4)).float()
c_next_data, h_next_data, output = lstm.forward(input_data, c_data, h_data)
print("New cell state c size:", c_next_data.shape)
print("New hidden state h size:", h_next_data.shape)
print("Predicted output y size:", output.shape)
```

```
New cell state c size: torch.Size([1, 4])
New hidden state h size: torch.Size([1, 4])
Predicted output y size: torch.Size([1, 1])
```

The PyTorch LSTM (equivalent to ours, but can be repeated num_layers times in a row)

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

        # LSTM cell
        self.lstm = torch.nn.LSTM(input_size, hidden_size, num_layers = self.num_layers, batch_first = True)

        # Linear layer for final prediction
        self.linear = torch.nn.Linear(hidden_size, output_size)

    def forward(self, inputs, cell_state, hidden_state):
        # Forward pass through the LSTM cell
        hidden = cell_state, hidden_state
        output, new_memory = self.lstm(inputs, hidden)
        cell_state, hidden_state = new_memory
        output = self.linear(output)
        return cell_state, hidden_state, output
```

Works and ready to be trained as in NB4!

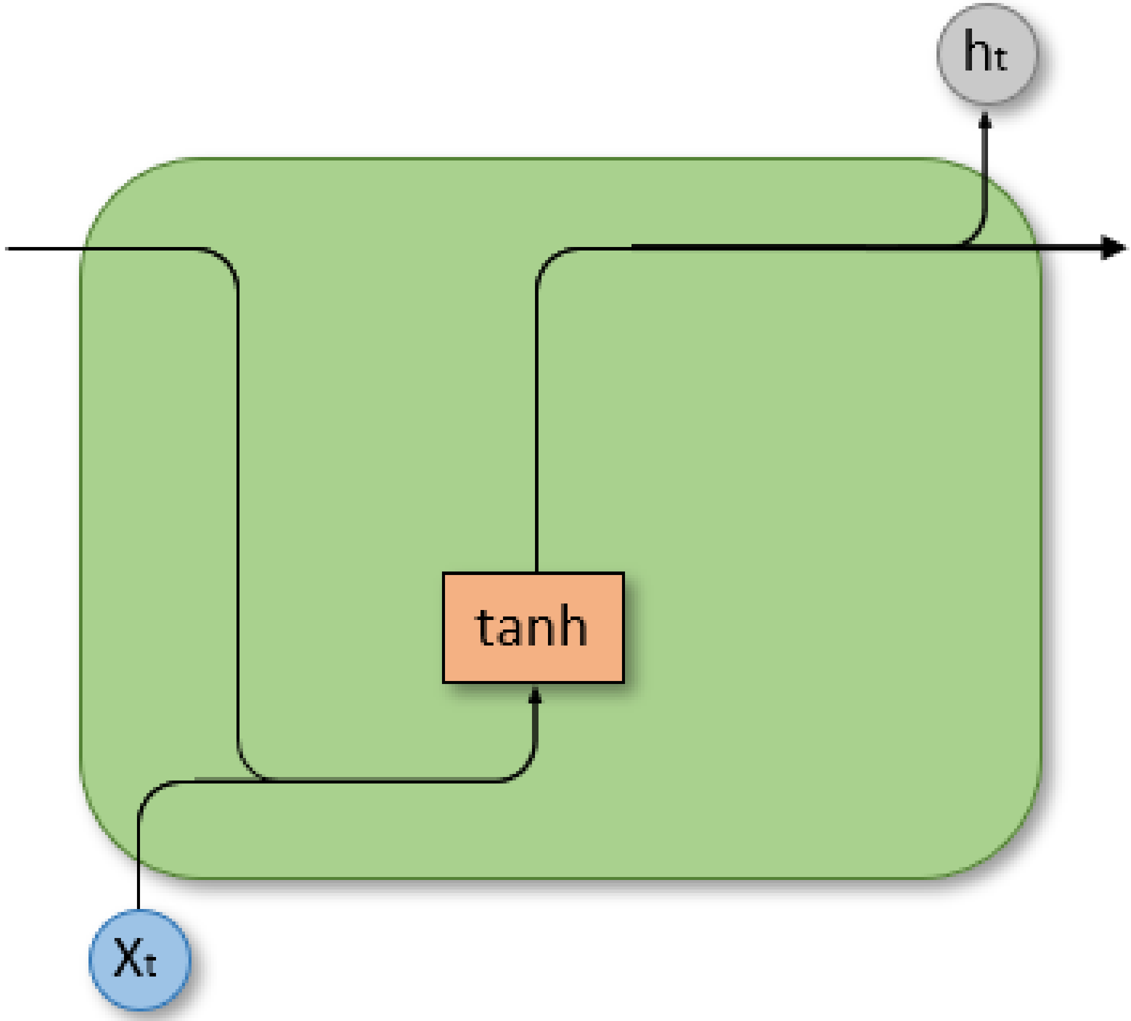
```
# Define the model parameters
input_size = 1
hidden_size = 4
num_layers = 1
output_size = 1

#Create the model
model = LSTM_pt(input_size, hidden_size, num_layers, output_size).to(device)
print(model)
```

```
LSTM_pt(
  (lstm): LSTM(1, 4, batch_first=True)
  (linear): Linear(in_features=4, out_features=1, bias=True)
)
```

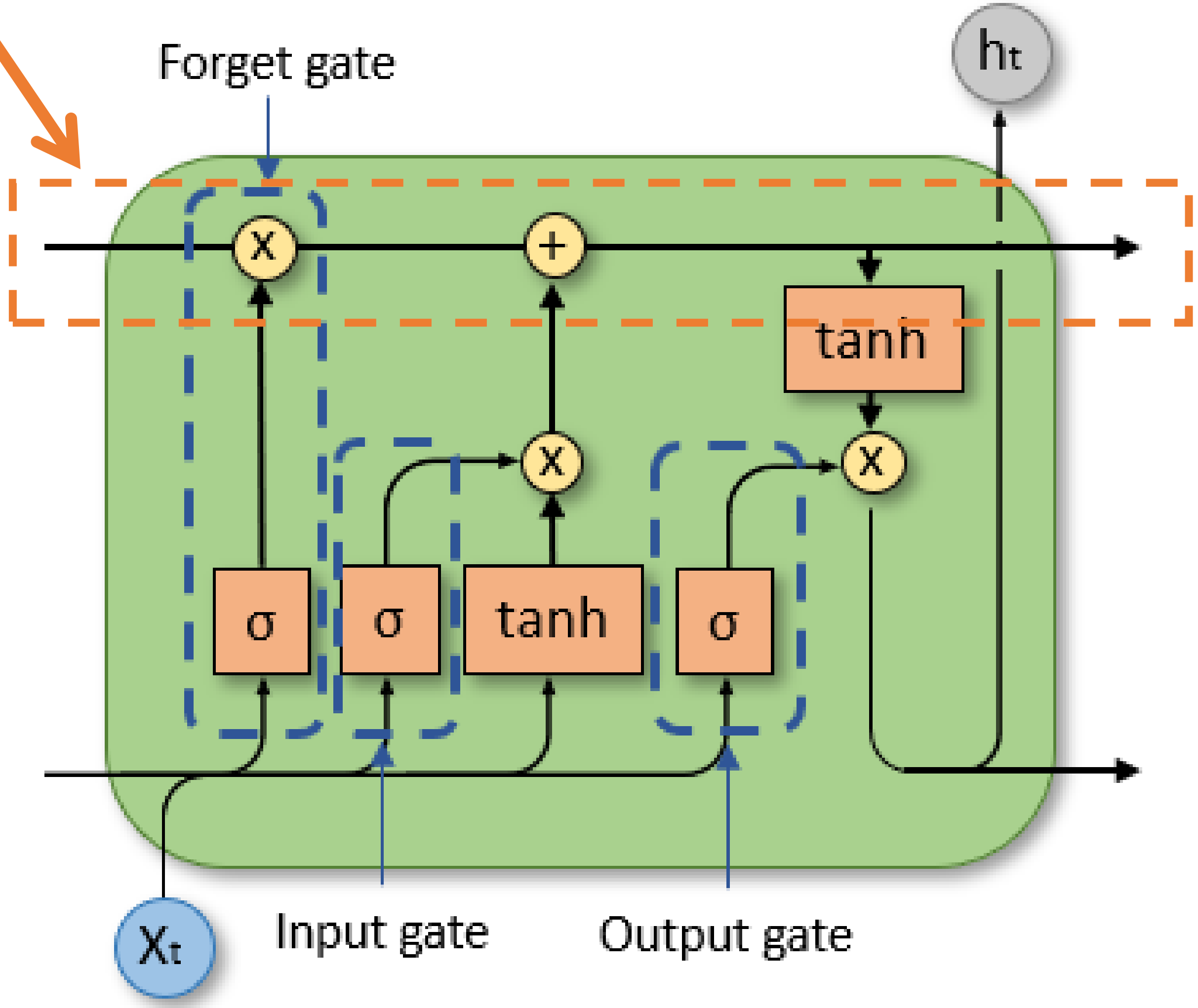
```
# Testing out GRU model
lstm = LSTM_pt(input_size, hidden_size, num_layers, output_size)
input_data = torch.from_numpy(np.random.randn(1, 1)).float()
h_data = torch.from_numpy(np.random.randn(1, 4)).float()
c_data = torch.from_numpy(np.random.randn(1, 4)).float()
c_next_data, h_next_data, output = lstm.forward(input_data, c_data, h_data)
print("New cell state c size:", c_next_data.shape)
print("New hidden state h size:", h_next_data.shape)
print("Predicted output y size:", output.shape)
```

```
New cell state c size: torch.Size([1, 4])
New hidden state h size: torch.Size([1, 4])
Predicted output y size: torch.Size([1, 1])
```



RNN

A very nice highway for information to backpropagate on rapidly, without any vanishing gradient effect!
(similar to a residual, but over time?)



LSTM

Introducing GRU

Definition (Gated Recurrent Units models):

Introduced in [Cho2014], Gated Recurrent Units **models (or** GRUs) propose to use gating mechanisms to selectively **update** and **discard** information in the hidden vector, implementing "**remember**" and "**forget**" operations of the brain.

It allows the model to selectively choose **which information from the previous hidden state to retain and which to discard**, in order to give more importance to recently predicted values, for instance.

In addition, this can help to maintain the gradients and avoid the vanishing gradient problem, and makes these models better suited for modeling complex time series data that exhibit long-term dependencies.

Introducing GRU

Below are the equations used in the forward method of a GRU model.

- **Update Gate:** combine input x_t and previous memory h_{t-1}, using two sets of weight matrices W_z and U_z, and one bias b_z.

$$z_t = \sigma(W_z\, x_t + U_z\, h_{t-1} + b_z)$$

The presence of a sigmoid, forces z_t to have a value in $[0, 1]$.
(Will be important later on).

- The update gate will later be used by the neural network to selectively decide how much information from the previous hidden vector should be forgotten or remembered, and how much should be replaced with a newly produced memory state.
- In a sense, it is similar to how the brain decide the proportion of information it should forget or replace over time.

Introducing GRU

Below are the equations used in the forward method of a GRU model.

- **Reset Gate:** Similar to the formula used in the update gate, but with different parameters W_r, U_r, b_r.

$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r)$$

The presence of a sigmoid, forces r_t to have a value in $[0, 1]$.
(Will be important later on).

- The reset gate will later be used by the neural network to selectively decide how much information from the previous hidden vector should be used to compute the new memory value.
- In a sense, it is similar to how the brain decide the proportion of information it should forget or remember over time.

Introducing GRU

Below are the equations used in the forward method of a GRU model.

- **Candidate Hidden State:** Define the potential new hidden state $\widetilde{h_t}$ that could replace h_{t-1} and become the next memory vector h_t.

$$\widetilde{h_t} = \tanh(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h)$$
$$(\odot \ \textit{is the element wise multiplication})$$

A value 0 in r_t means that the previous hidden state will be entirely discarded, whereas a value 1 means that the previous hidden state will be entirely kept.

Introducing GRU

Below are the equations used in the forward method of a GRU model.

- **New Hidden State:** Define the new hidden state h_t that will replace h_{t-1}, by combining a proportion of the old memory state $h_{t-1}$and a proportion of the candidate hidden state $\widetilde{h_t}$.

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \widetilde{h_t}$$
$$(\odot \ is\ the\ element\ wise\ multiplication)$$

A value 1 in z_t means that the new hidden state will be entirely based on the candidate hidden vector, whereas a value 0 means that the new hidden state will be entirely based on the previous hidden one.

Introducing GRU

Below are the equations used in the forward method of a GRU model.

- **Predicted Output:** Finally, use the new hidden vector to make a prediction y_t that hopefully matches the value of x_{t+1}.

Defined as a simple linear operation, as before in LSTM.

$$y_t = Linear(h_t)$$

To recap, our GRU

x_t

h_{t-1}

GRU Neural Network prediction at time t

$$
\begin{aligned}
z_t &= \sigma(W_z\, x_t + U_z\, h_{t-1} + b_z) \\
r_t &= \sigma(W_r x_t + U_r h_{t-1} + b_r) \\
\widetilde{h_t} &= \tanh(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h) \\
h_t &= (1 - z_t) \odot h_{t-1} + z_t \odot \widetilde{h_t} \\
y_t &= Linear(h_t)
\end{aligned}
$$

y_t (pred. for x_{t+1})

h_t

Building our own GRU

Let us start by defining all the weights and biases we need for each of the 5 operations we discussed.

We implement them in the init method of our class as before.

Pay attention to the sizes used for each parameter.

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

        # Parameters for the reset gate
        self.Wr = torch.nn.Parameter(torch.randn(input_size, hidden_size))
        self.Ur = torch.nn.Parameter(torch.randn(hidden_size, hidden_size))
        self.br = torch.nn.Parameter(torch.zeros(hidden_size))
        # Parameters for the update gate
        self.Wz = torch.nn.Parameter(torch.randn(input_size, hidden_size))
        self.Uz = torch.nn.Parameter(torch.randn(hidden_size, hidden_size))
        self.bz = torch.nn.Parameter(torch.zeros(hidden_size))
        # Parameters for the candidate hidden state
        self.W = torch.nn.Parameter(torch.randn(input_size, hidden_size))
        self.U = torch.nn.Parameter(torch.randn(hidden_size, hidden_size))
        self.b = torch.nn.Parameter(torch.zeros(hidden_size))

        # Parameters for the output prediction
        self.V = torch.nn.Parameter(torch.randn(hidden_size, output_size))
        self.b = torch.nn.Parameter(torch.randn(1, output_size))
```

Building our own GRU

Eventually, compute all 5 GRU operations in the forward method.

```
def forward(self, inputs, hidden):
    # Compute the reset gate
    reset_gate = torch.sigmoid(torch.matmul(inputs, self.Wr) + torch.matmul(hidden, self.Ur) + self.br)
    # Compute the update gate
    update_gate = torch.sigmoid(torch.matmul(inputs, self.Wz) + torch.matmul(hidden, self.Uz) + self.bz)

    # Compute the candidate hidden state
    candidate_hidden = torch.tanh(torch.matmul(inputs, self.W) + torch.matmul(reset_gate * hidden, self.U) + self.b)
    # Compute the updated hidden state
    new_hidden = (1 - update_gate) * hidden + update_gate * candidate_hidden

    # Compute the output
    output = torch.matmul(new_hidden, self.V) + self.b

    return new_hidden, output
```

Building our own GRU

We can then quickly check it has the expected behaviour, with consistency on the sizes.

```
# Testing out GRU model
gru = GRU(input_size = 1, hidden_size = 4, output_size = 1)
input_data = torch.from_numpy(np.random.randn(1, 1)).float()
h_data = torch.from_numpy(np.random.randn(1, 4)).float()
h_next_data = gru.forward(input_data, h_data)
print("New hidden state h size:", h_next_data[0].shape)
print("Predicted output y size:", h_next_data[1].shape)
```

```
New hidden state h size: torch.Size([1, 4])
Predicted output y size: torch.Size([1, 1])
```

The PyTorch GRU (equivalent to ours, but can be repeated num_layers times in a row)

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

        # GRU cell
        self.gru = torch.nn.GRU(input_size, hidden_size, num_layers = 1, batch_first = True)
        # Output layer
        self.linear = torch.nn.Linear(hidden_size, output_size)

    def forward(self, inputs, hidden):
        # Forward pass through the GRU cell
        out, new_hidden = self.gru(inputs, hidden)
        # Apply the output layer
        output = self.linear(out)
        return new_hidden, output
```

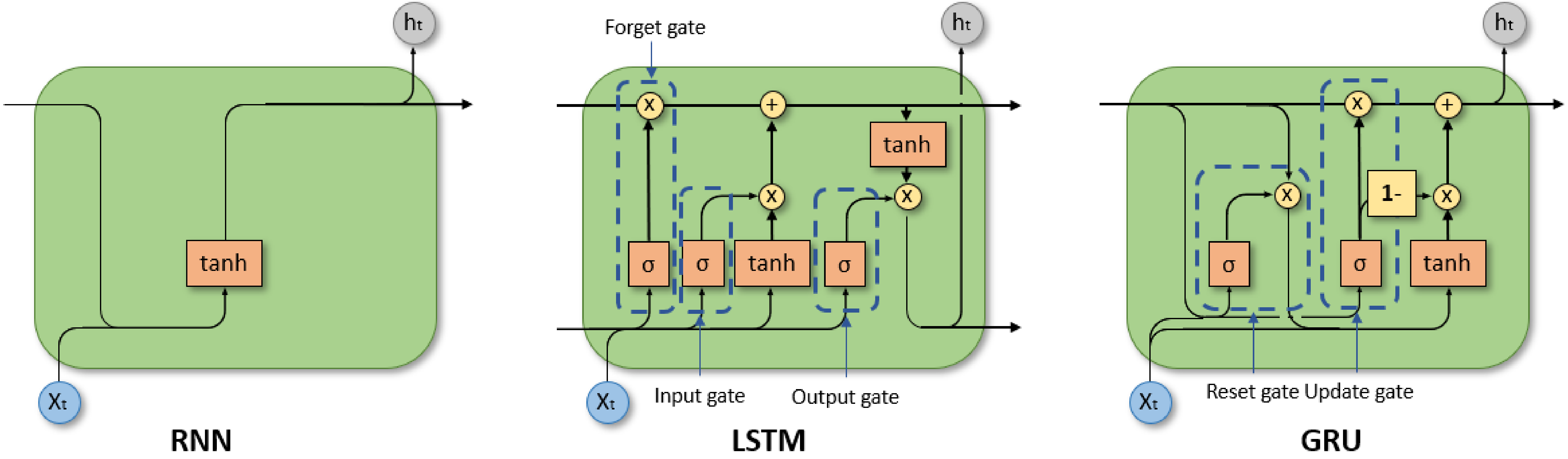
Works and ready to be trained as in NB5!

```
# Define the model parameters
input_size = 1
hidden_size = 16
num_layers = 1
output_size = 1
# Create the model
model = GRU_pt(input_size, hidden_size, num_layers, output_size).to(device)
print(model)
```

```
GRU_pt(
  (gru): GRU(1, 16, batch_first=True)
)
```

```
# Testing out GRU model
gru = GRU_pt(input_size = 1, hidden_size = 4, num_layers = 1, output_size = 1)
input_data = torch.from_numpy(np.random.randn(1, 1)).float()
h_data = torch.from_numpy(np.random.randn(1, 4)).float()
h_next_data = gru.forward(input_data, h_data)
print("New hidden state h size:", h_next_data[0].shape)
print("Predicted output y size:", h_next_data[1].shape)
```

```
New hidden state h size: torch.Size([1, 4])
Predicted output y size: torch.Size([1, 4])
```



ht
tanh
Xt
RNN
Forget gate
ht
x
+
tanh
x
x
σ
σ
tanh
σ
Xt
Input gate
Output gate
LSTM
ht
x
+
x
1-
x
σ
σ
tanh
Xt
Reset gate Update gate
GRU

LSTMs vs. GRUs

Both LSTM and GRU networks are effective at capturing long-term and short-term dependencies in time series data.

So, which one should we use?

There are some benefits to using LSTMs over GRUs:

- LSTMs are designed to maintain and propagate information over longer time lags than GRUs. This makes them better suited for tasks that require the network to retain information for longer periods of time, such as language modelling or speech recognition.

LSTMs vs. GRUs

There are some benefits to using LSTMs over GRUs:

- LSTMs have more parameters than GRUs, which can make them more expressive and better able to model complex nonlinear functions. This can be beneficial in tasks that require a high level of accuracy, such as image or speech recognition.
- LSTMs can handle input sequences of variable lengths more effectively than GRUs, as they have an explicit memory cell that can store information over multiple timesteps. This can be useful in applications where the length of the input sequence may vary, such as natural language processing.

LSTMs vs. GRUs

Of course, there are also some benefits to using GRUs over LSTMs:

- GRUs have fewer parameters than LSTMs, which can make them faster to train and less prone to overfitting.
- GRUs have a simpler structure than LSTMs, which can make them easier to implement and understand.
- GRUs can be more effective than LSTMs in handling sequences that have a lot of noise or missing data, as they are better able to adapt to changes in the input. They are in fact less prone to overfitting, given that they have an architecture with lower complexity

LSTMs vs. GRUs

Of course, there are also some benefits to using GRUs over LSTMs:

- GRUs can be more computationally efficient than LSTMs, as they have fewer computations per time step.
- GRUs are better suited for tasks that require the network to prioritize recently observed information over older information. This is because GRUs have a gating mechanism that is designed to selectively update and discard information in the hidden state, which makes them better able to model short-term dependencies.

Learn more about these topics

Out of class, supporting papers, for those of you who are curious.

- [Hochreiter1997] S. **Hochreiter** and J. **Schmidhuber** “Long short-term memory”, 1997.
- [Gers2013] F. A. Gers, J. **Schmidhuber**, F. Cummin, “Learning to forget: Continual prediction with LSTM”, 2013.
- [Cho2014] K. **Cho**, B. van Merrienboer, C. Gulcehre, D. **Bahdanau**, F. Bougares, H. Schwenk, and Y. Bengio, “Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation”, 2014.

Learn more about these topics

Tracking important names (Track their works and follow them on Scholar, Twitter, or whatever works for you!)

- **Kyunghyun Cho: Professor** at **New York University**.
  https://scholar.google.co.uk/citations?user=0RAmmIAAAAAJ&hl=en
  https://kyunghyuncho.me/
- **Dzmitry Bahdanau: Research Scientist** at **Element AI** and **Adjunct Professor** at **McGill University**.
  https://scholar.google.de/citations?user=Nq0dVMcAAAAJ&hl=en
  https://rizar.github.io/

Learn more about these topics

Tracking important names (Track their works and follow them on Scholar, Twitter, or whatever works for you!)

- **Jürgen Schmidhuber: Professor** at the **University of Lugano in Switzerland**.
  https://scholar.google.com/citations?user=gLnCTgIAAAAJ&hl=fr
  https://people.idsia.ch/~juergen/
- **Sepp Hochreiter: Professor** at the **Johannes Kepler University of Linz in Austria**.
  https://scholar.google.at/citations?user=tvUH3WMAAAAJ&hl=en
  https://www.iarai.ac.at/people/sepphochreiter/

Learn more about these topics

Some extra (easy) reading and videos for those of you who are curious.

- [LesleyEdu] “How Memories Are Made: Stages of Memory Formation”, explains in simple terms how the human brain processes information and generates memory.
  https://lesley.edu/article/stages-of-memory
- [Medium] “GPT-3, RNNs and All That: A Deep Dive into Language Modelling”, explains how Large Language Models (such as ChatGPT) work using autoregressive RNNs.
  Note: the transformer part will be covered on Week 8!
  https://towardsdatascience.com/gpt-3-rnns-and-all-that-deep-dive-into-language-modelling-7f67658ba0d5