50.039 Theory and Practice of Deep Learning

W6-S2 Times Series Data and Recurrent Neural Networks

Matthieu De Mari

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

About this week (Week 6)

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 6)

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)

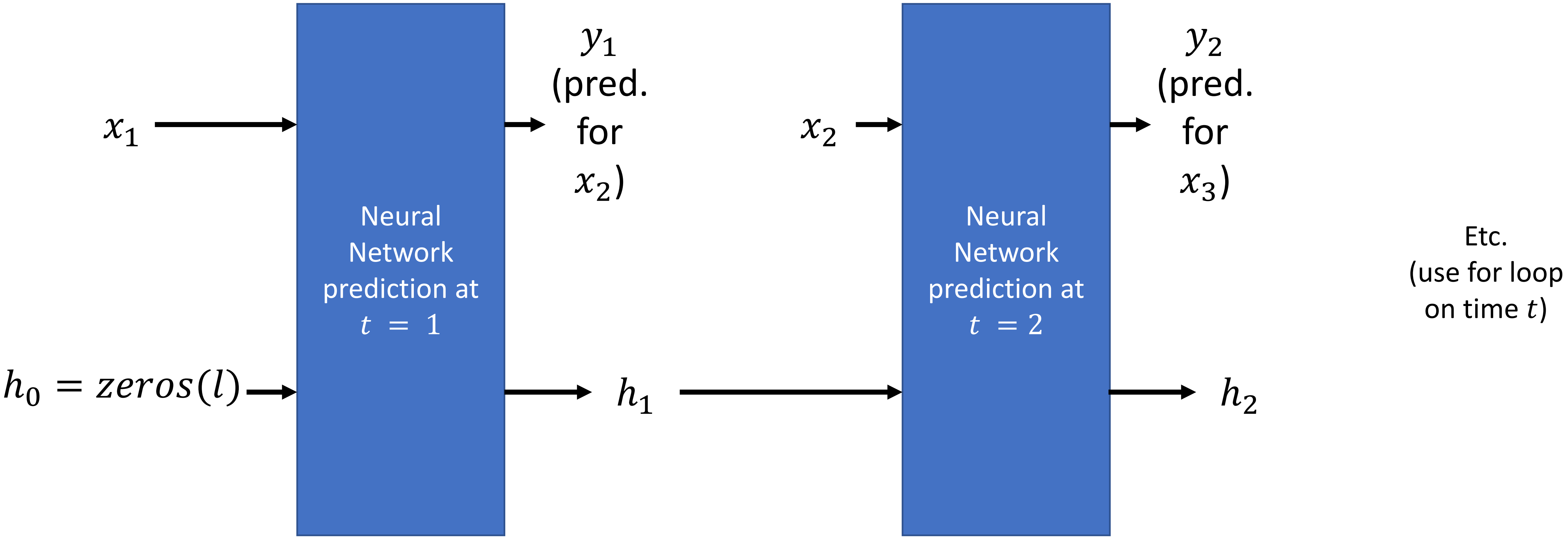
Definition (Recurrent Neural Network):

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

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

```
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

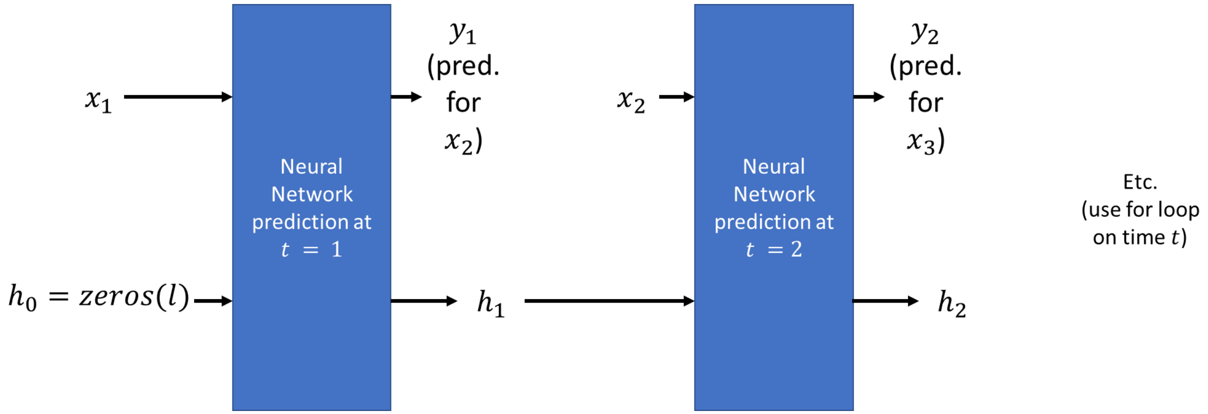
Definition (backpropagation through time):

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.

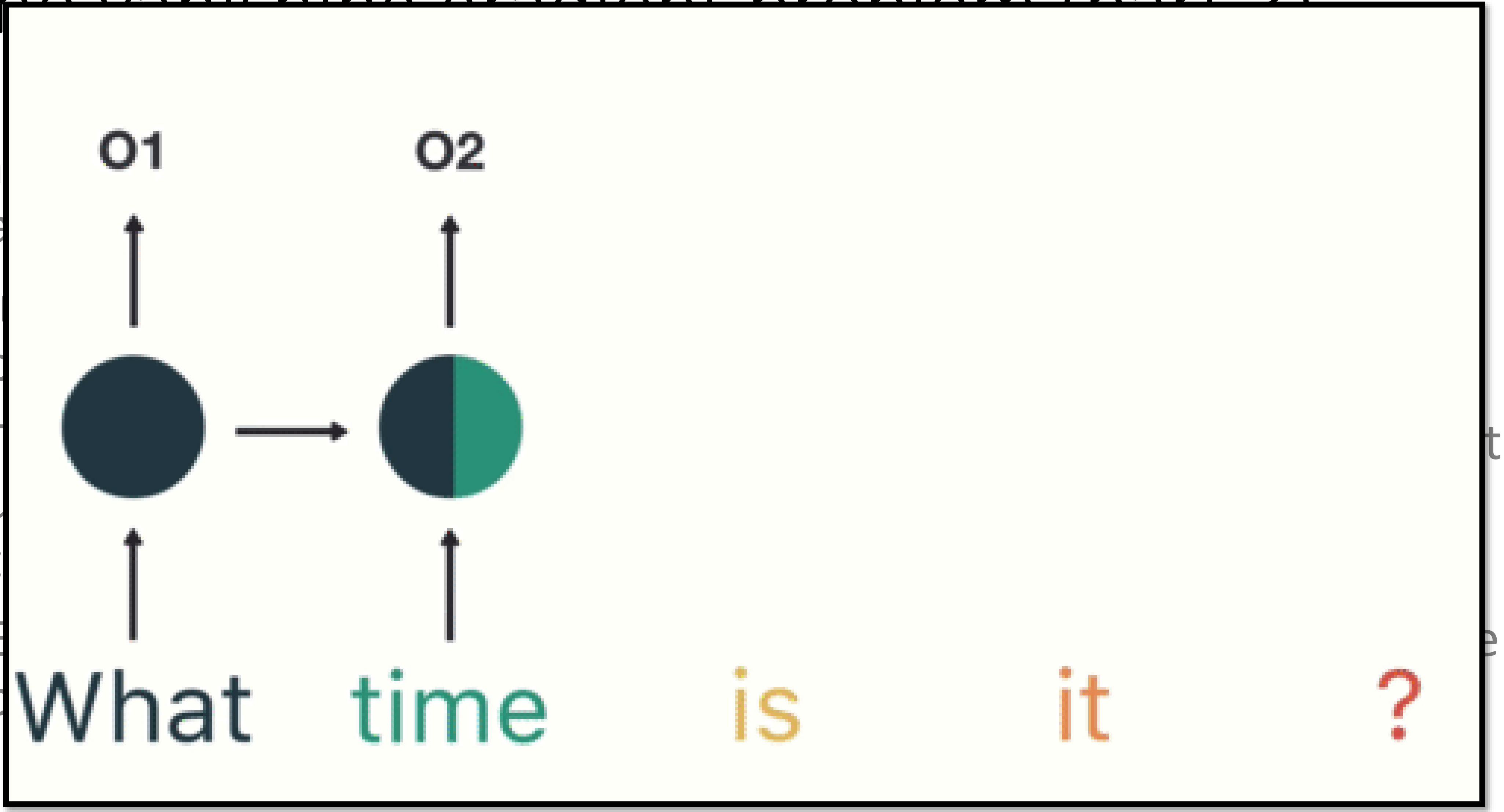


The vanishing gradient problem (part 3)

The **vanishing gradient problem** can often occur in RNNs during training.

- In RNNs, each hidden state h_{t+1} is updated via a function of the previous hidden state h_t and the input x_t.
- The memory vector will attempt to integrate "new" information, but ultimately will have to discard "old" information to make space for the "new" information.
- Eventually "old" information will be lost over time, which could have proven useful in some predictions…

The vanishing gradient problem (part 2)



The vanishing gradient problem (part 3)

The **vanishing gradient problem** can often occur in RNNs during training.

- In RNNs, each hidden state h_{t+1} is updated via a function of the previous hidden state h_t and the input x_t.
- During training, we used backpropagation over time, learning how to update the memory vectors over time for that problem/dataset.
- However, **this backpropagation can result in very small gradients for the earlier time steps**, where the weights are then updated very slowly.

The vanishing gradient problem (part 3)

The **vanishing gradient problem** can often occur in RNNs during training.

- This can be particularly problematic for RNNs, since they are designed to handle time series data that can exhibit long-term dependencies.
- The vanishing gradient problem can be particularly severe in RNNs that use activation functions that saturate, such as the hyperbolic tangent (tanh) or sigmoid functions.
- Indeed, when the activation function saturates, the gradients become very small, making it difficult to propagate information through the network.

The vanishing gradient problem (part 3)

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.**
- Therefore, 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 things that happened over time.
- **Or, in other words, the cognitive ability of <u>memorizing and processing information</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?

- Brain 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

Core idea behind GRU and LSTM models

How to improve our current vanilla RNN (shown):

- Have **distinct sets of trainable parameters** for the neural network (one set for each task),
- And **different sets of operations happening in the forward method** (again, one set for each task).
- **Decorrelate both tasks!**

```
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
```

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**.
- 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} (hence, “remembering” the previous state),
- A new input, decided as a function of the previous hidden state h_{t-1} and the current observation x_t, eventually weighted by $i_t \in [0, 1]$.

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: but, wait, 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 "memory" for the LSTM and 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**.
- 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 kind of "working 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 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:

$$y_t = o_t h_t$$

It simply combines the output gate with the new hidden state vector.

It can also be defined as a simple linear operation with a weight V:

$$y_t = V h_t$$

To recap, our LSTM

x_t

h_{t-1}, c_{t-1}

LSTM Neural Network prediction at time t

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

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

h_t, c_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 fc each parameter.

```
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 output prediction
        self.V = torch.nn.Parameter(torch.randn(hidden_size, output_size))
```

Restricted

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 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 output gate
    output_gate = torch.sigmoid(torch.matmul(inputs, self.Wo) + torch.matmul(hidden_state, self.Uo) + self.bo)

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

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

    return cell_state, hidden_state, output
```

Building our own LSTM

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

```python
# 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

        # LSTM cell
        self.lstm = torch.nn.LSTM(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, cell_state, hidden_state):
        # Forward pass through the LSTM cell
        hidden = cell_state, hidden_state
        output, new_hidden = self.lstm(inputs, hidden)
        cell_state, hidden_state = hidden

        # Apply the output layer to the last timestep of the output sequence
        output = self.linear(output)

        return cell_state, hidden_state, output
```

Works and ready to be trained as in NB3!

```
# 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])
```

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 $\widetilde{h_t}$.

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

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 with a weight matrix V.

Most of the time, no bias will be included in this final operation.

$$y_t = V h_t$$

To recap, our GRU

x_t → GRU Neural Network prediction at time t → y_t (pred. for x_{t+1})

h_{t-1} → → h_t

GRU Neural Network prediction at time t

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

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))
```

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)

    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)

```python
class GRU_pt(torch.nn.Module):
    def __init__(self, input_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
        # 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 NB3!

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

```
GRU_pt(
  (gru): GRU(1, 16, batch_first=True)
  (linear): Linear(in_features=16, out_features=1, bias=True)
)
```

```python
# Testing out GRU model
gru = GRU_pt(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])
```

LSTMs vs. GRUs

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

There are however 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 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

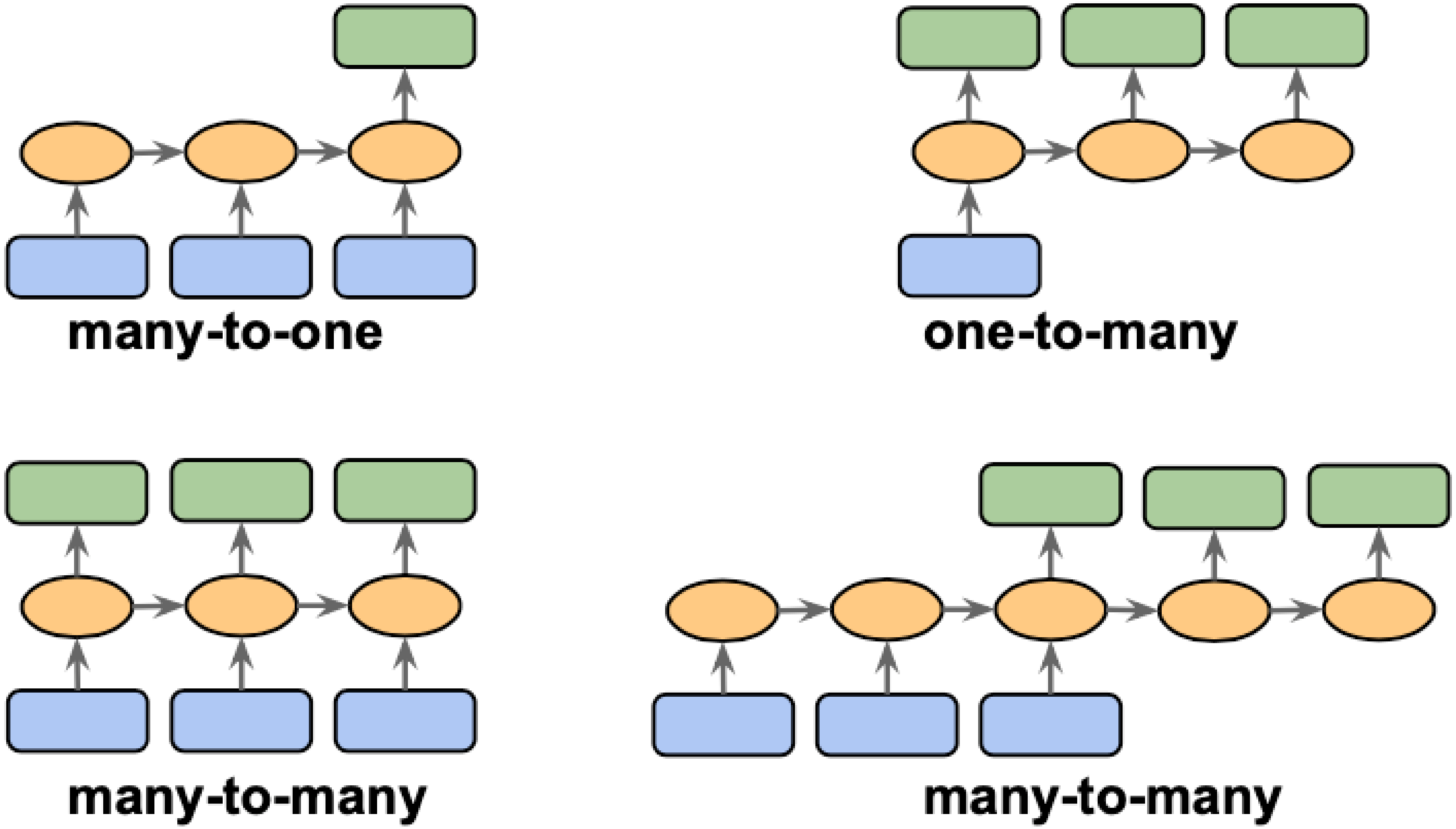
Of course, there are also some benefits to using GRUs over LSTMs:

- 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.
- 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.

A teaser on what is coming on W8

From RNNs to Natural Language Processing?

A quick word on one/many-to-one/many models



One-to-one models

Definition (one-to-one models):

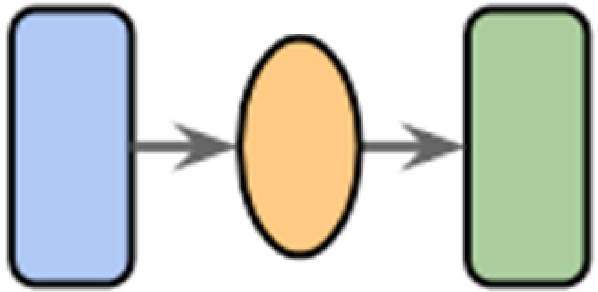
A **one-to-one model** is a machine learning model that:

- Takes **a single input**,
- And, optionally, **a memory vector**,
- And attempts to produce **a single output**.

This can be seen as the simplest type of machine learning problem.

Our problem, which attempted to predict:

- The value of a single point, x_{t+1},
- Given the value of a single point, being the current observation x_t,
- Along with a memory vector, h_t.



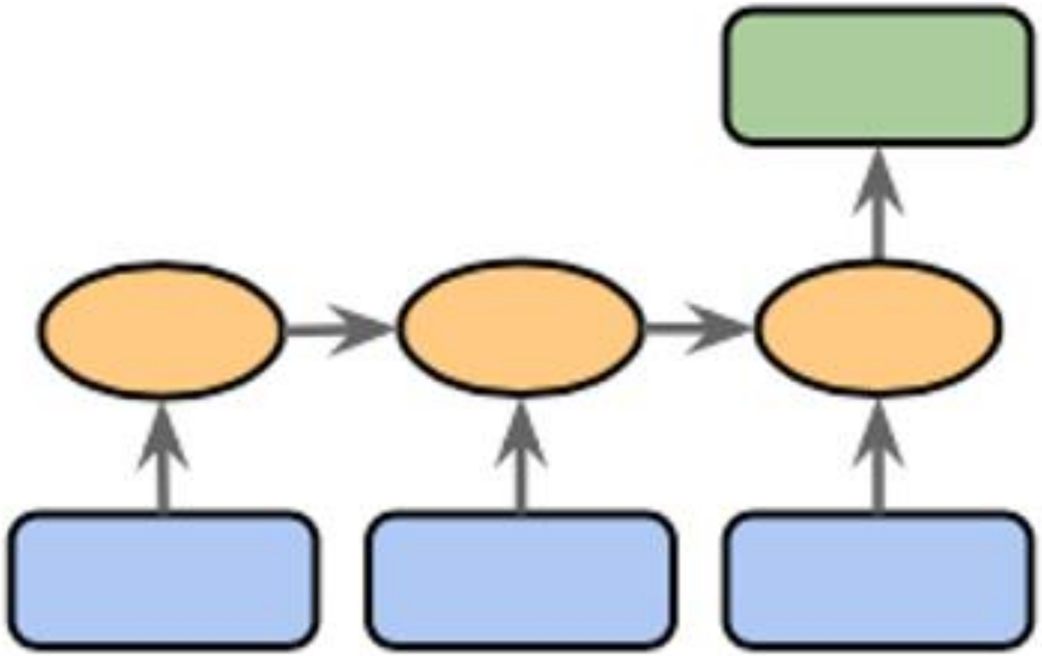
Many-to-one models

Definition (many-to-one models):

A **many-to-one model** is a machine learning model that:

- Takes **<u>many</u> inputs (in sequence or not)**,
- And, optionally, **a memory vector**,
- And attempts to produce **a single output**.

This is another common type of ML problem we have seen so far.



Many-to-one models

Definition (many-to-one models):

A **many-to-one model** is a machine learning model that:

- Takes **many inputs (in sequence or not)**,
- And, optionally, **a memory vector**,
- And attempts to produce **a single output**.

This is another common type of ML problem we have seen so far.

For instance,

- Our Notebook 2, using a history of several values as inputs to predict the next one.
- A next-word predictor, which attempts to predict the next word in a given sentence **(sentence = sequence of words ≈ time series!)**.

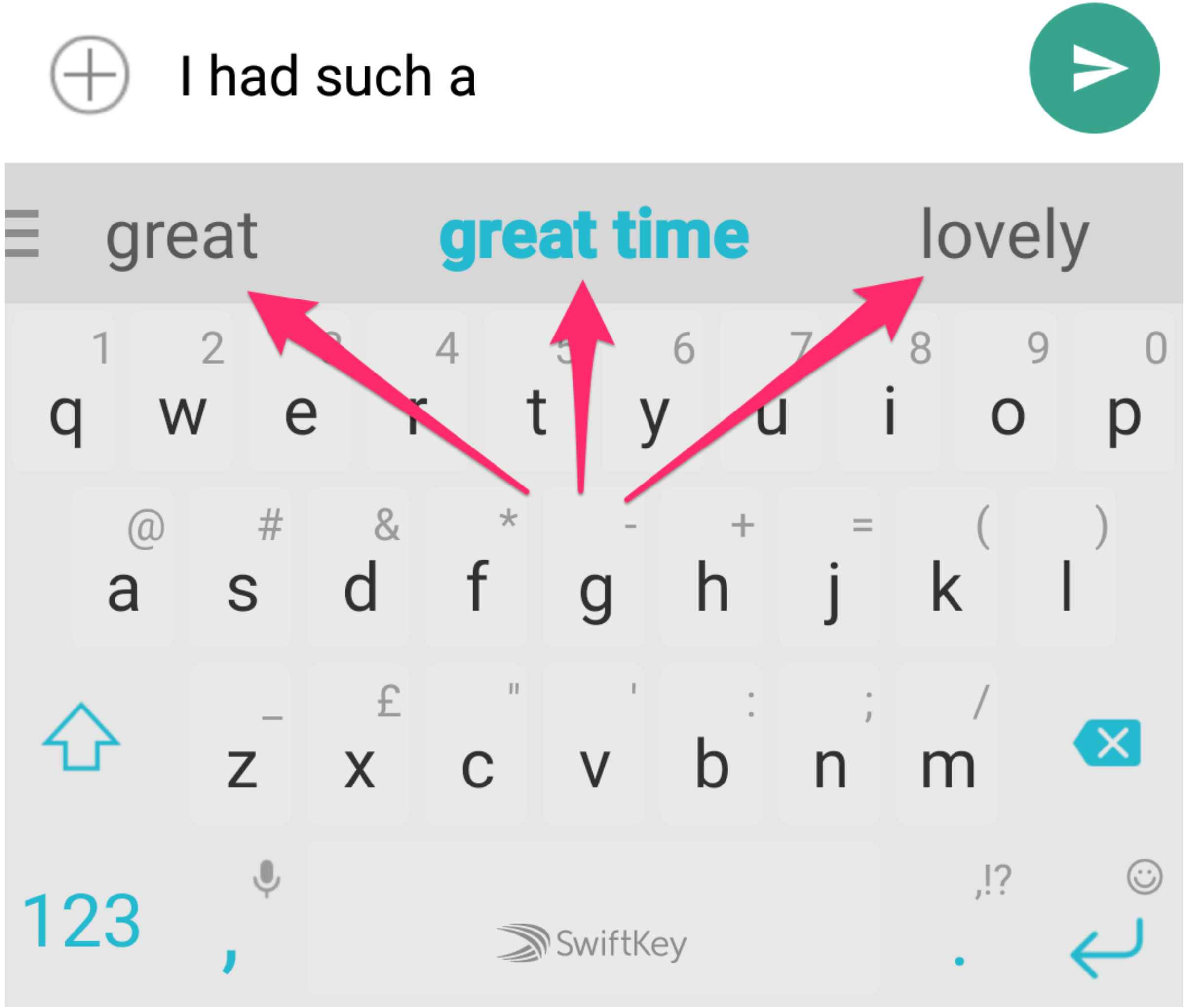
Many-to-one models

Definition (many-to-one models):

A **many-to-one model** is a machine learning model that:

- Takes **__many__ inputs (in sequence or not)**,
- And, optionally, **a memory vector**,
- And attempts to produce **a single output**.

This is another common type of ML problem we have seen so far.



Many-to-one models

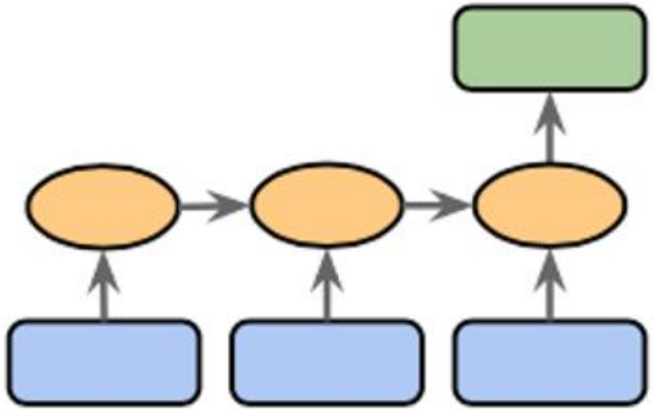
Definition (many-to-one models):

A **many-to-one model** is a machine learning model that:

- Takes **many inputs (in sequence or not)**,
- And, optionally, **a memory vector**,
- And attempts to produce **a single output**.

This is another common type of ML problem we have seen so far.

It would be possible to use a RNN of some sort for these problems, and only keep the final output (discarding all other calculated in the process), to compute a loss function of some sort.



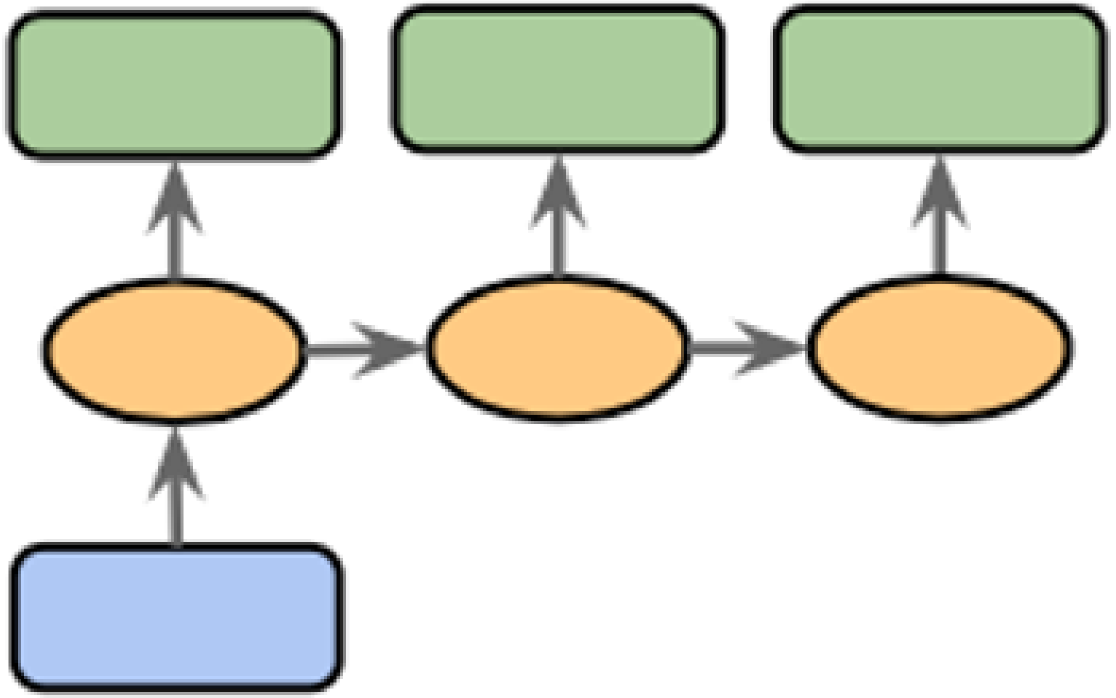
One-to-many models

Definition (one-to-many models):

A **one-to-many model** is…

You get what it means now.

To be honest, very few problems fall in the category of one-to-many, because most of the time we prefer to have a higher dimensionality on inputs than outputs…



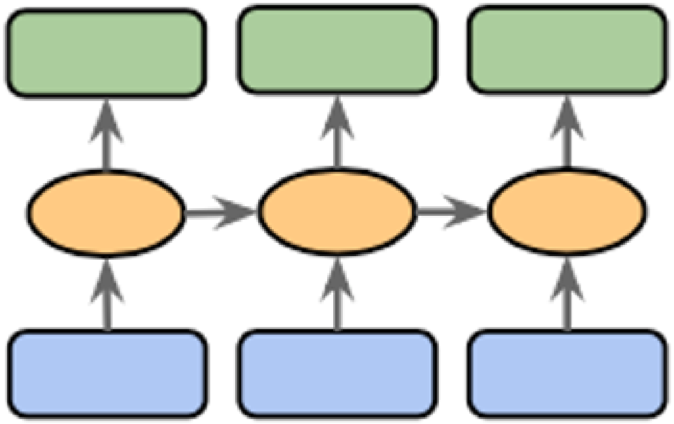
Many-to-many models

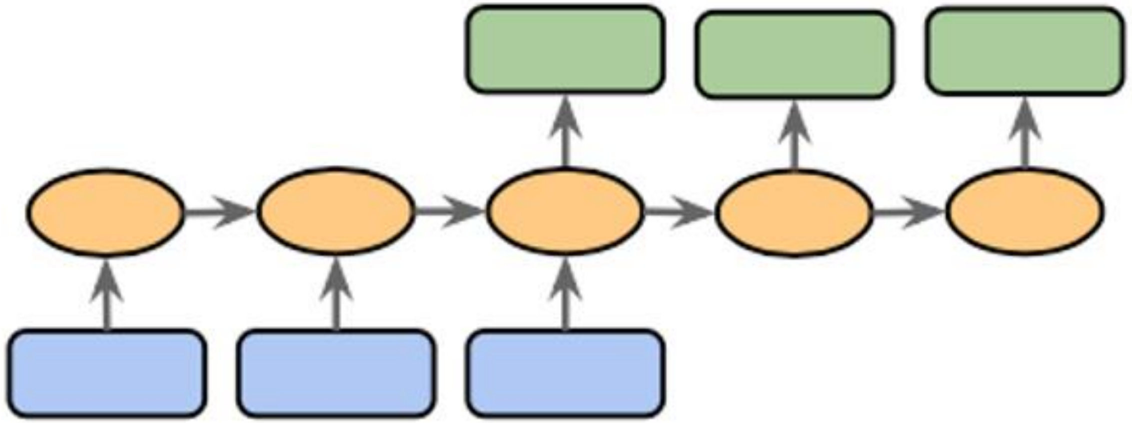
Definition (many-to-many models):

A **many-to-many model** is a machine learning model that:

- Takes **many inputs (in sequence or not)**,
- And, optionally, **a memory vector**,
- And attempts to produce **many outputs**.

This is also a very common type of ML problems, which we will investigate more on the coming weeks.





Many-to-many models

Definition (many-to-many models):

A **many-to-many model** is a machine learning model that:

- Takes **many inputs (in sequence or not)**,
- And, optionally, **a memory vector**,
- And attempts to produce **many outputs**.

This is also a very common type of ML problems, which we will investigate more on the coming weeks.

For instance,

- A stock market predictor that takes several points as inputs (history) and attempts to predict the next few values of the stock prices instead of just the next one.
- A translation model that translates text from English to French.

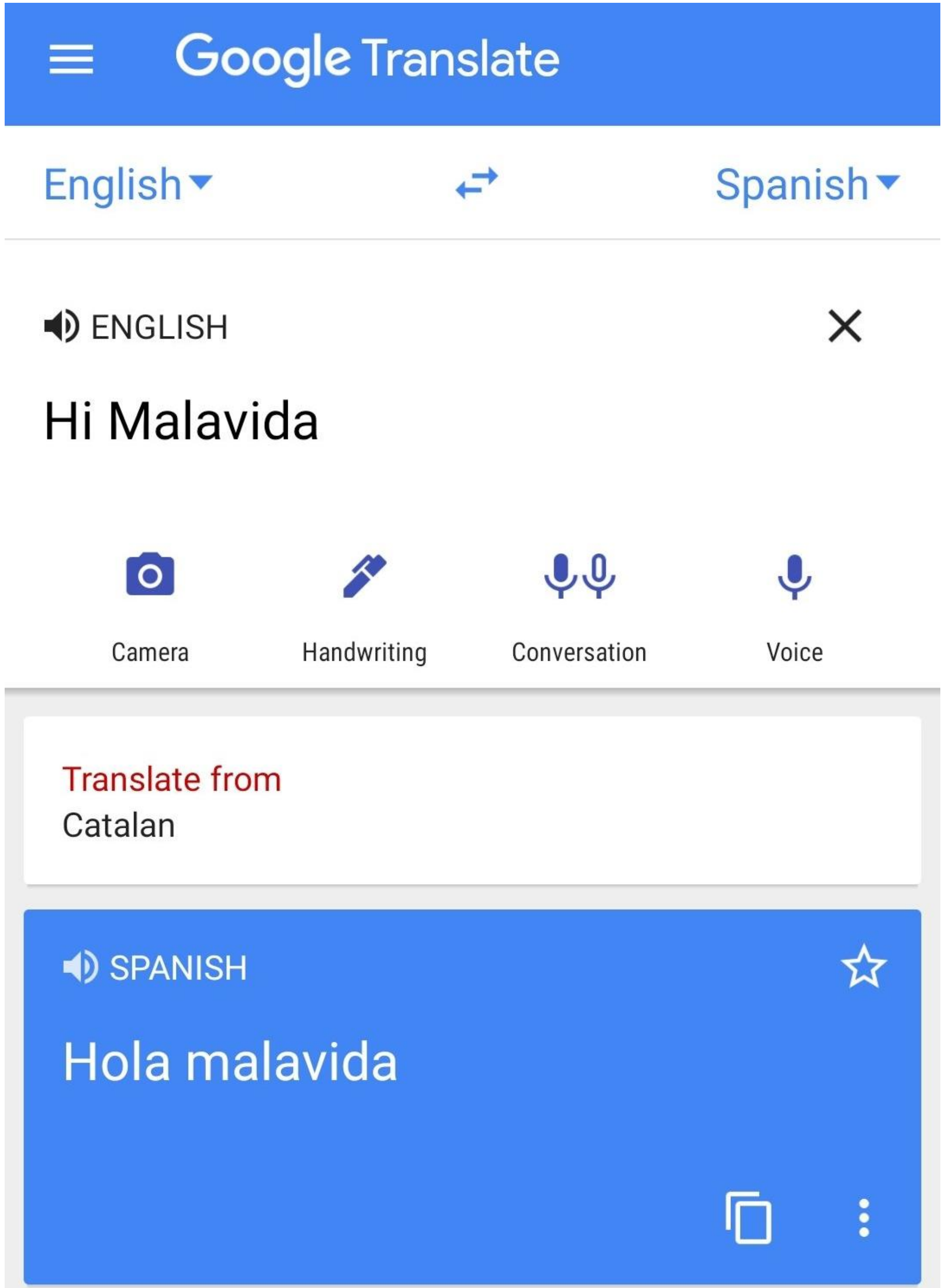
Many-to-many models

Definition (many-to-many models):

A **many-to-many model** is a machine learning model that:

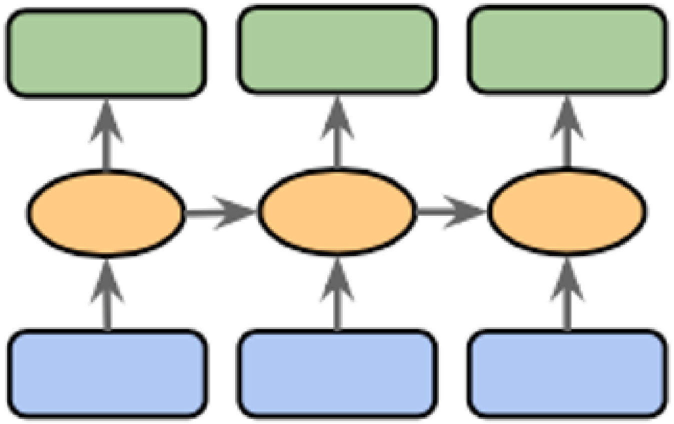
- Takes **many inputs (in sequence or not)**,
- And, optionally, **a memory vector**,
- And attempts to produce **many outputs**.

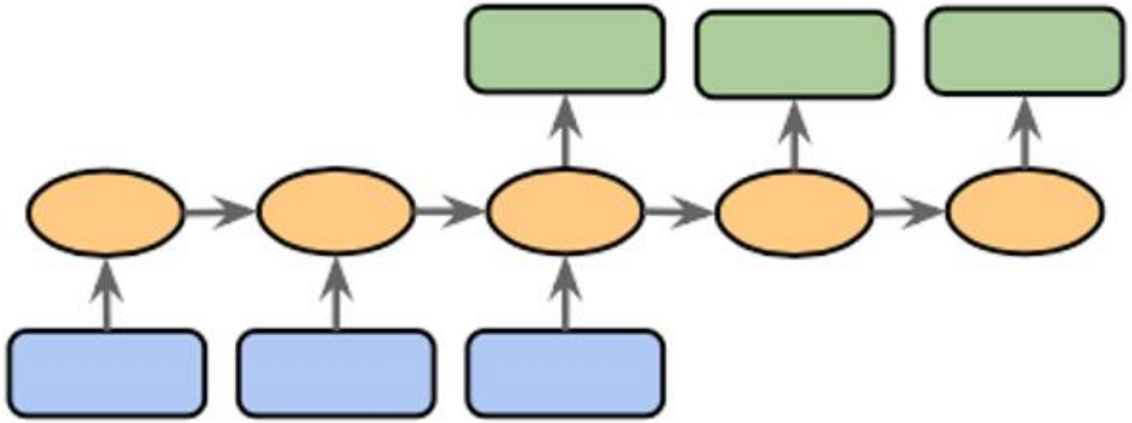
This is also a very common type of ML problems, which we will investigate more on the coming weeks.



Many-to-many models

→ But then, what is the difference between these two diagrams?





Seq2Seq models

Definition (Seq2Seq models):

A **Sequence-to-Sequence (Seq2Seq) model** is a special type of many-to-many model, where the inputs and outputs being produced are both **sequences**.

An apartment predictor with multiple inputs features, predicting the price of selling and the rental price as outputs, is not **Seq2Seq,** but it is **many-to-many**!

But our previous examples are **Seq2Seq**:

- A stock market predictor that takes several points as inputs (history) and attempts to predict the next few values of the stock prices instead of one.
- A translation model that translates text from English to French.

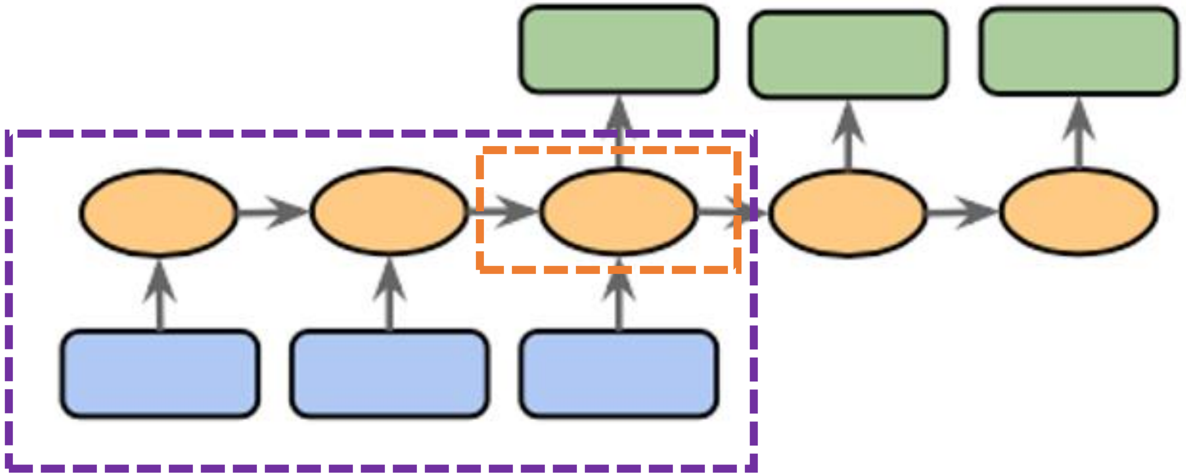
Encoder and Decoder models

Definition (encoder and decoder models):

In general, the first part of a **Seq2Seq** model will focus on analysing the input sequence, eventually producing a final **memory vector** output after having seen all inputs. **This is called an encoder.**

Can be done with a RNN of some sort, where the final memory state is called the **encoding vector**.

E.g. read an English sentence, progressively encoding each word's meaning, to obtain the entire sentence meaning in a single encoding vector.



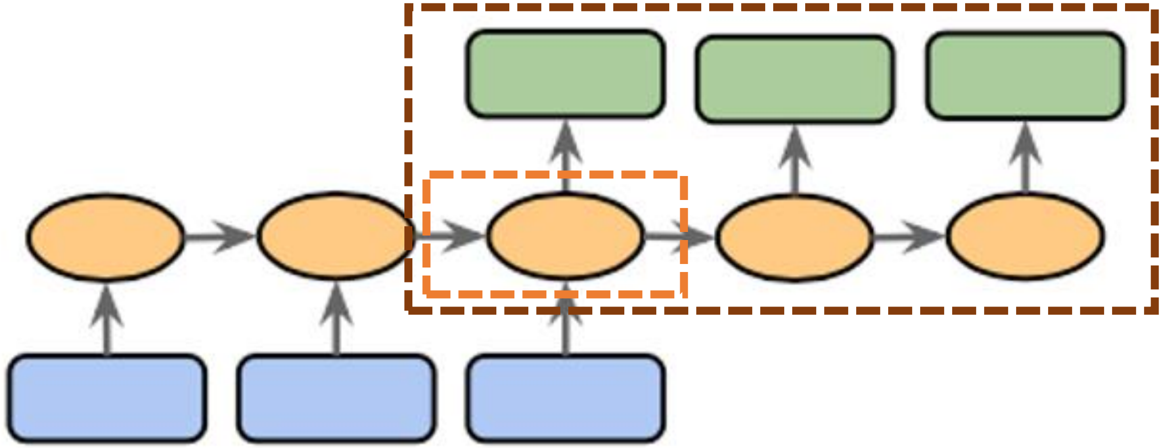
Encoder and Decoder models

Definition (encoder and decoder models):

In general, the second part of the model will focus on analysing the produced encoding vector, and producing a sequence of some sort as output. **This is called a decoder.**

Can be done with the same (or another) RNN of some sort, where the first memory state consists of the encoding vector.

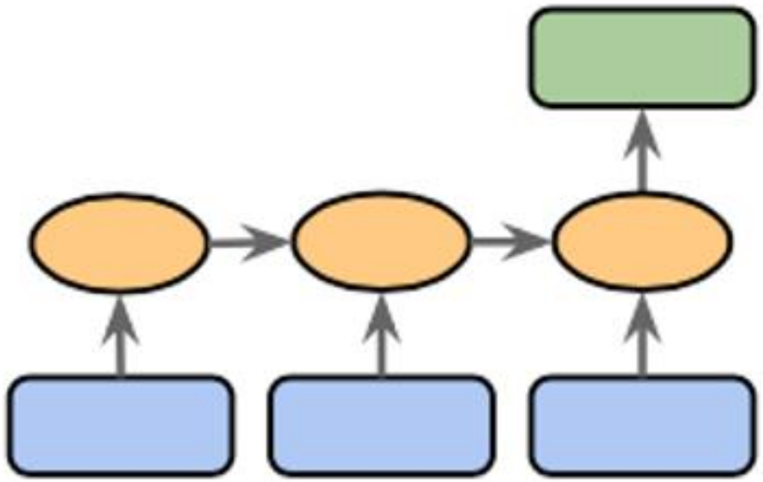
E.g. read an encoding vector describing an English sentence and produce an output sequence being the French translation.

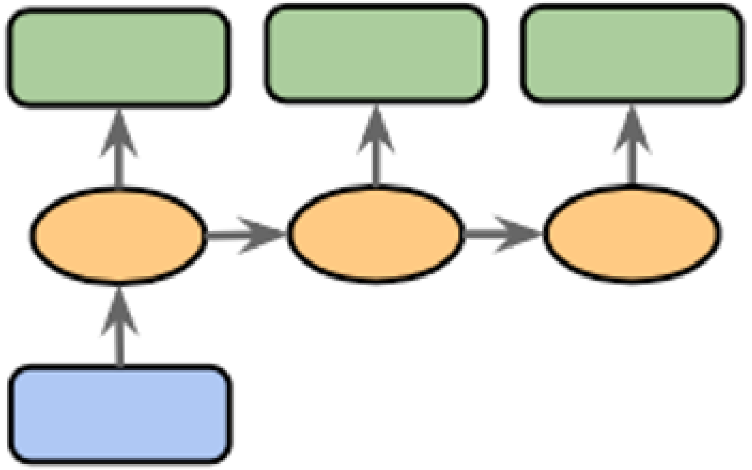


Encoder and Decoder models

Encoder model: can be seen as a many-to-one model of some sort, receiving a sequence as input and producing a single **encoding vector** as output.

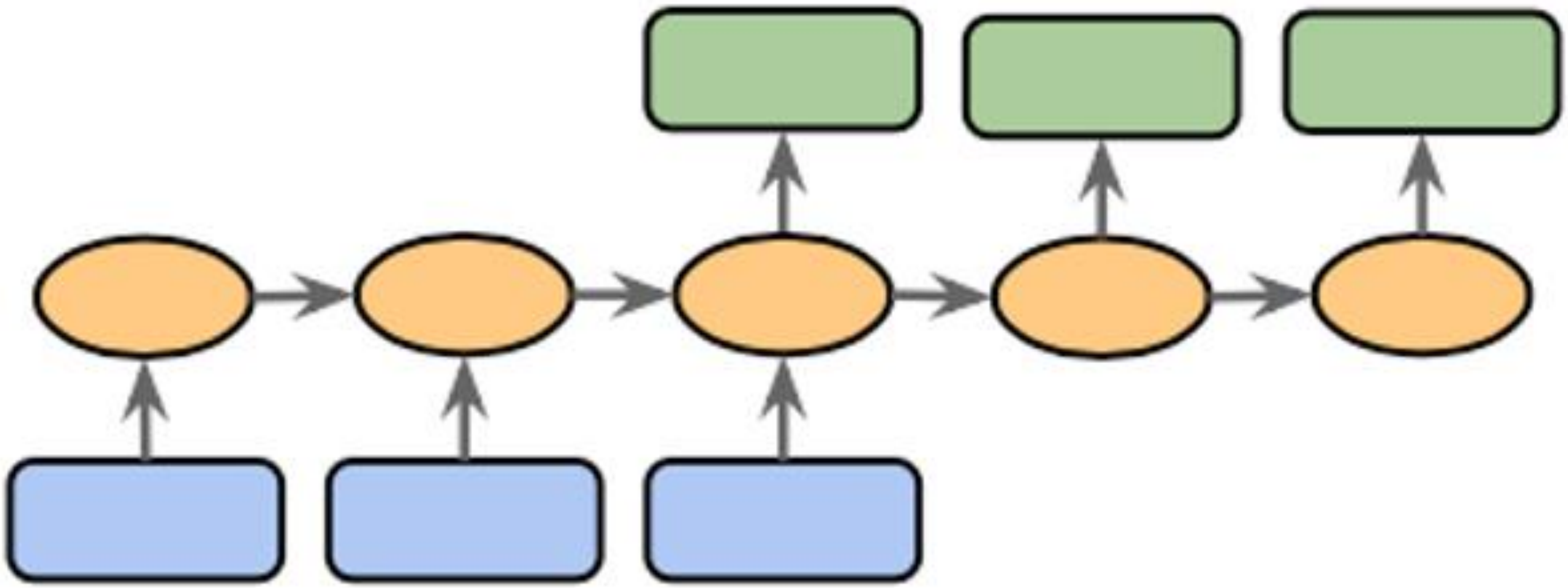
Decoder model: does the opposite (kind of), receiving this single **encoding vector** as input and producing a sequence as output.





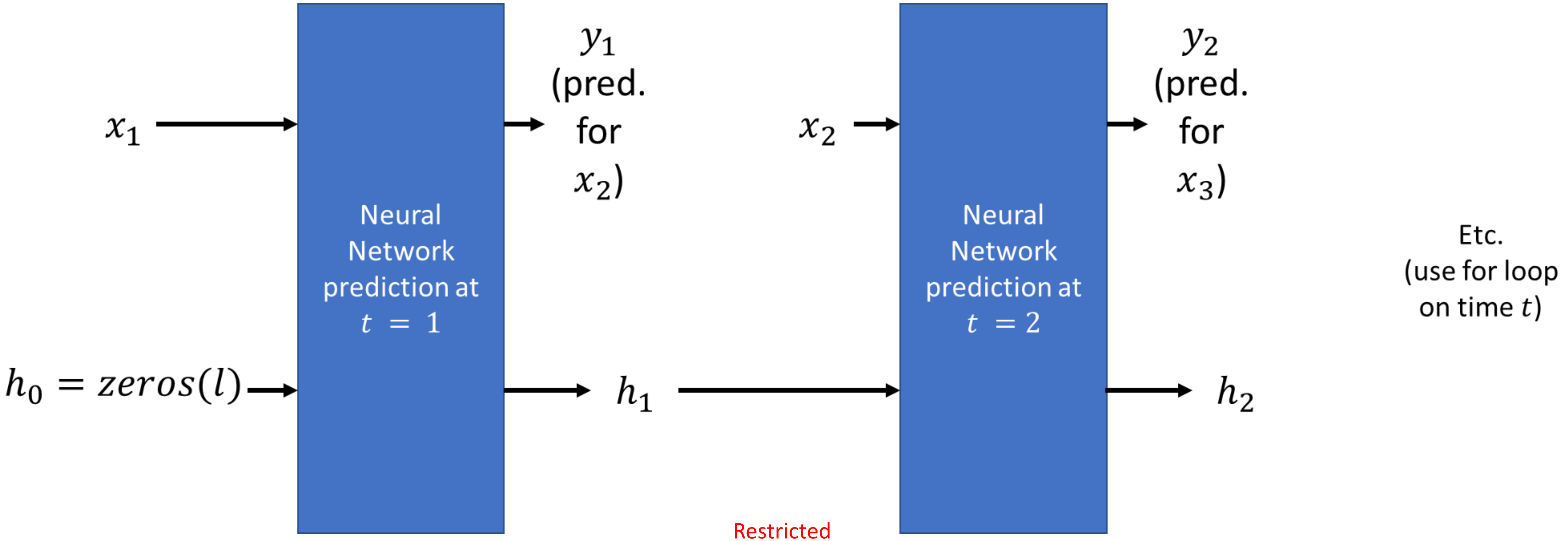
Encoder + Decoder = Seq2Seq?

Eventually both the **encoder** and the **decoder** models will assemble into a full **Seq2Seq** model.



A quick word on autoregressive RNNs

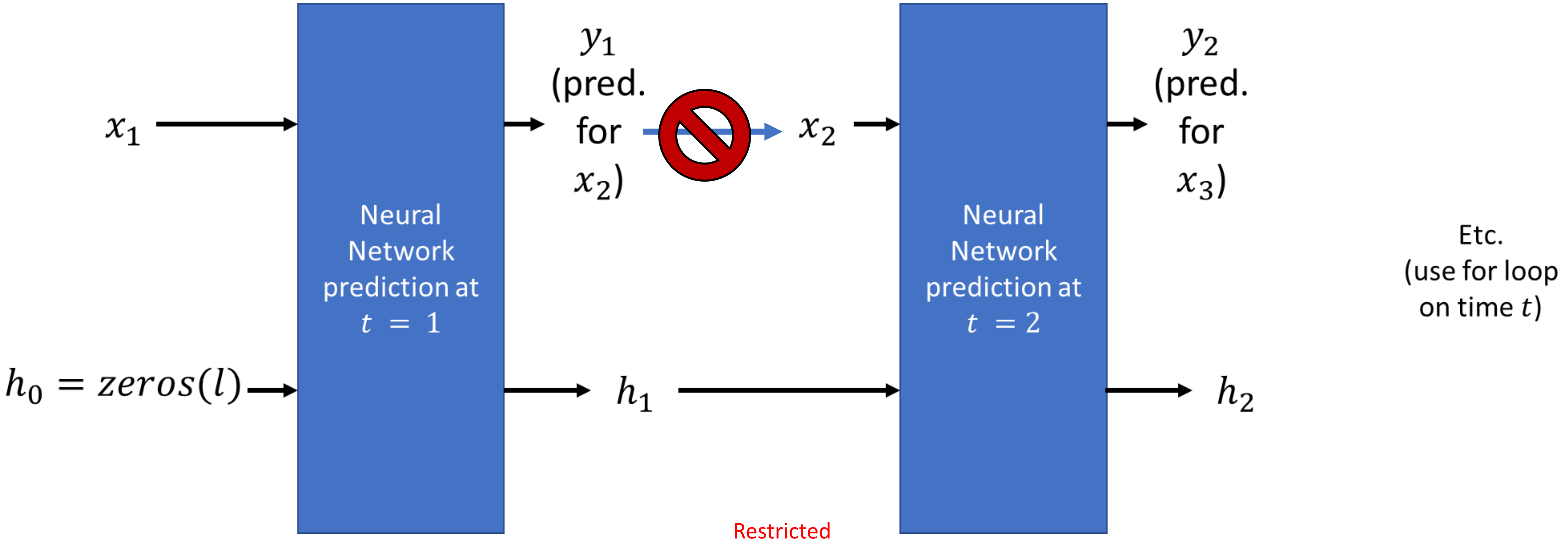
Our previous RNN models looked like this: start with an empty memory vector h_0, predict y_t and update memory vector h_t **using a new observation $\boldsymbol{x_t}$** as input on each time step.



Restricted

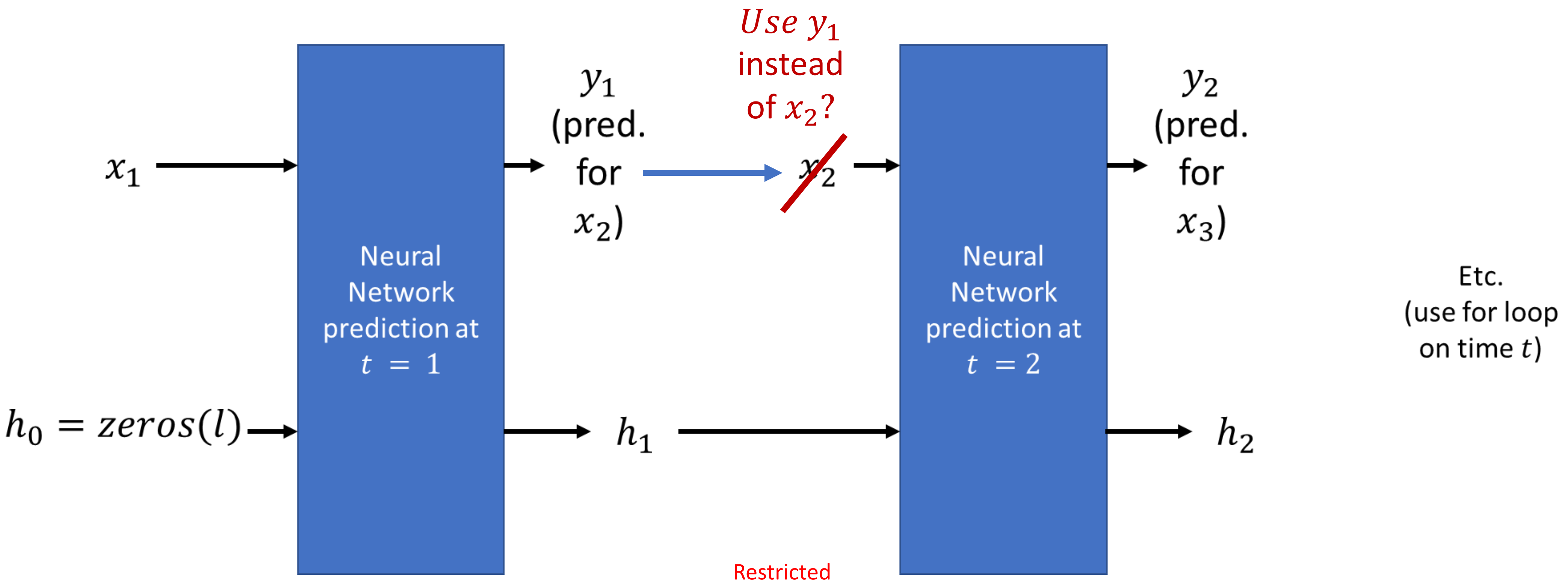
A quick word on autoregressive RNNs

Important: our RNNs models would not be using our prediction y_t as the new observation for the next step x_{t+1}!



A quick word on autoregressive RNNs

But what if we did?

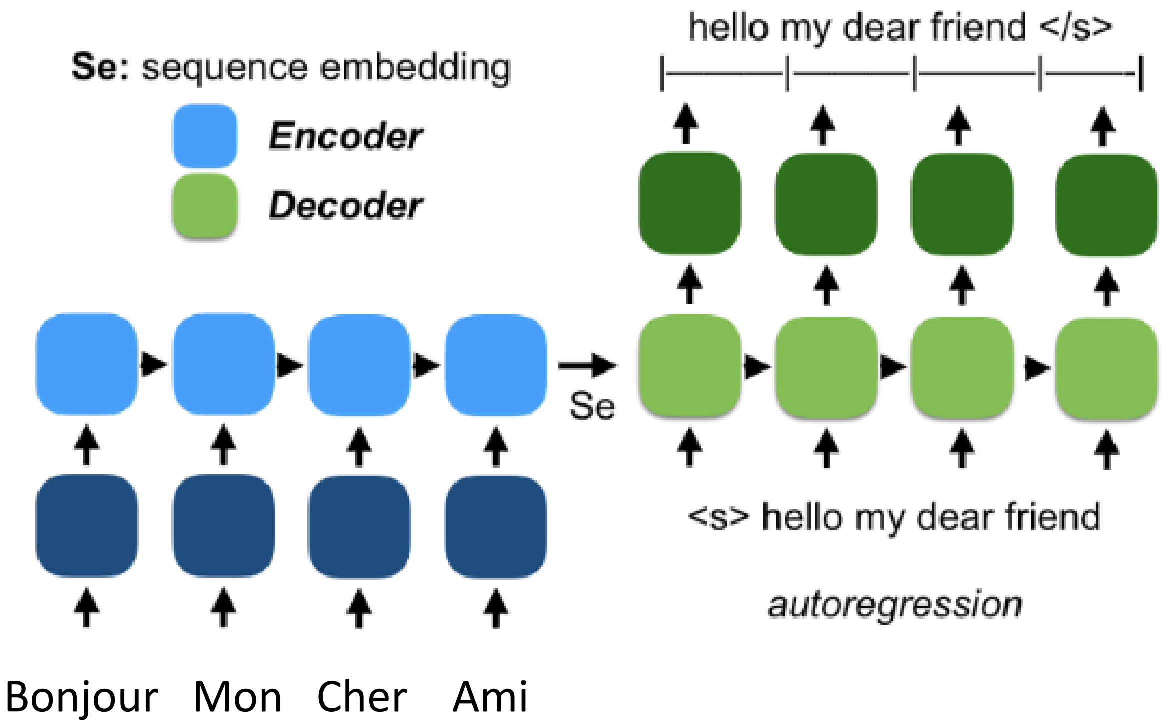


A quick word on autoregressive RNNs

Definition (autoregressive RNNs):

An **autoregressive** **RNN**, is a special type of RNN, which **reuses the previous predictions in place of the next observation to make the next prediction**. Basically, using y_t in place of x_{t+1}.

This is usually harder to train, as you would only use the "real" observed value x_1 at $t = 1$. After that use your own predictions y_t in place of x_{t+1}!

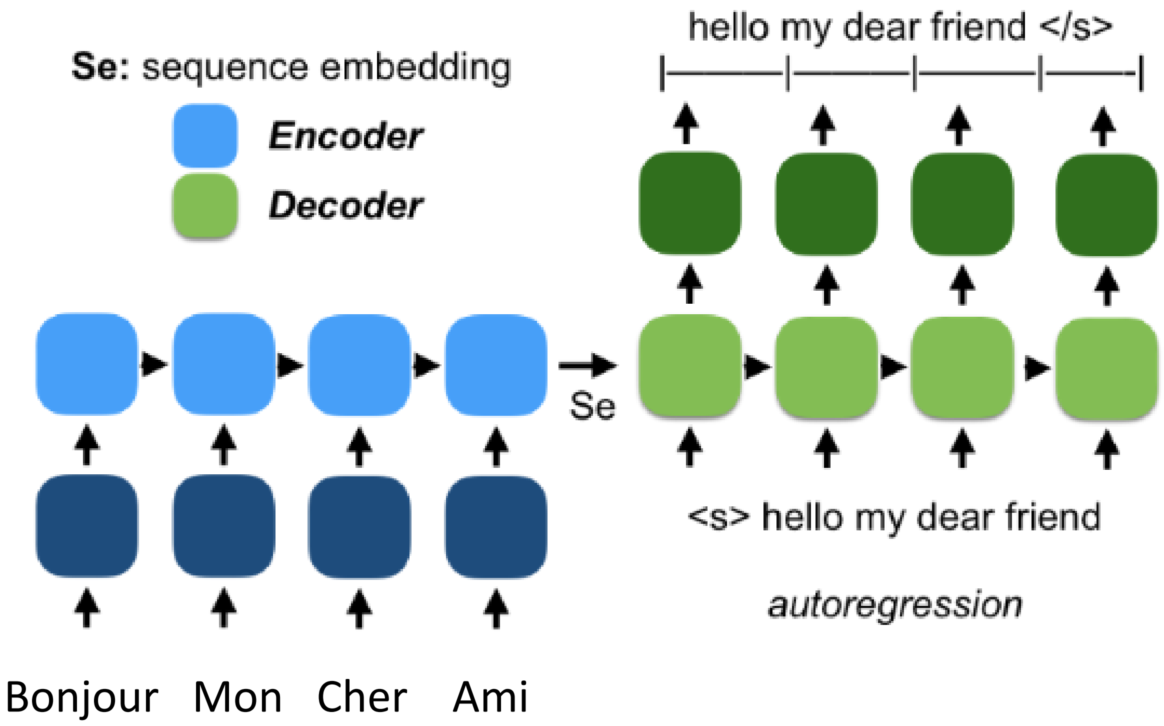


Restricted

A quick word on autoregressive RNNs

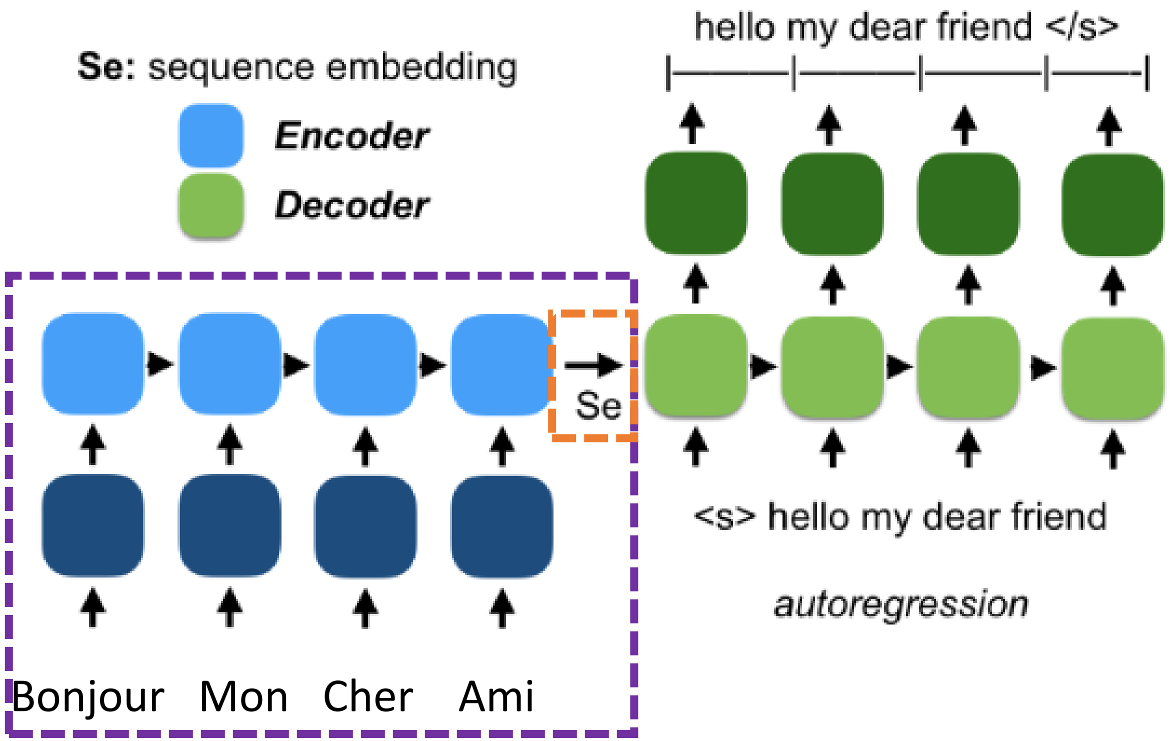
Typically, we love to use these autoregressive models in Natural Language Processing, for instance when it comes to generating output sequences.

E.g. generate an English sentence for a given encoding vector describing the meaning of a French sentence.



A quick word on autoregressive RNNs

Encoder reads the French sentence and produces a **sequence embedding/encoding vector** ***Se*** as the final memory vector.

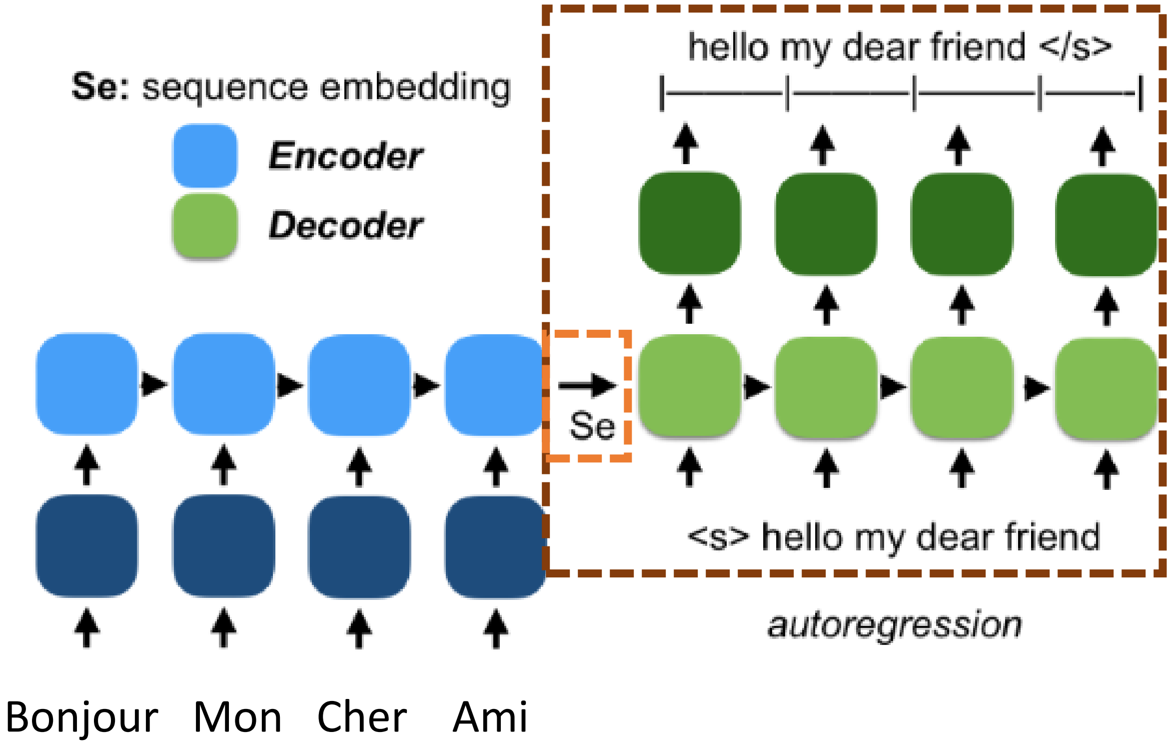


A quick word on autoregressive RNNs

Decoder part receives this **encoding vector** as the initial memory vector value h_0 and the first value x_1 is $< s >$, a special word indicating the beginning of a sequence.

The first word *"hello"* is produced as y_1 and reused as x_2, to produce the next word *"my"*.

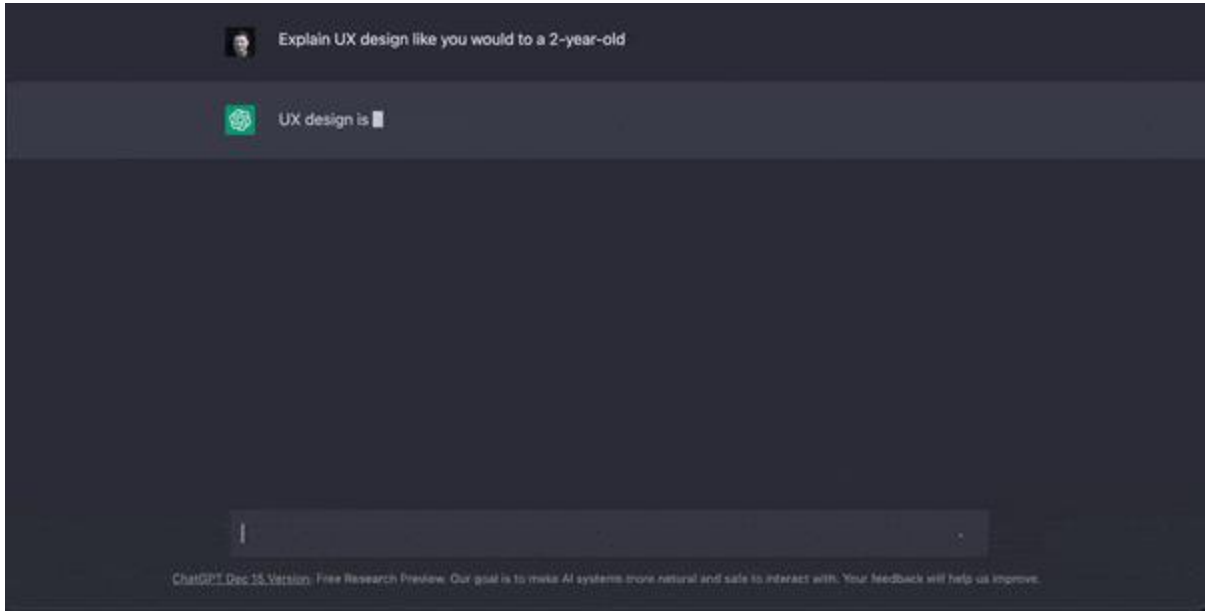
Until eventually, another special word $</s>$ is produced, indicating the end of the sentence.



A quick word on autoregressive RNNs

Typically, **ChatGPT** is a Seq2Seq model using autoregressive RNNs!

- Takes a sequence of words as input (the user question) and **encodes** it,
- **And reconstructs an answer, a sentence, one word after the other!**



A quick word on autoregressive RNNs

Typically, **ChatGPT** is a Seq2Seq model using autoregressive RNNs!

- Takes a sequence of words as input (the user question) and **encodes** it,
- **And reconstructs an answer, a sentence, one word after the other!**

We will learn more about these during Weeks 8 and 12!

(And during the NLP specialty course on Term 7!)

Conclusion (Week 6)

- Time series datasets
- Deep learning models with history
- History length tradeoffs
- Implementing memory in Deep Learning models
- Our first vanilla Recurrent Neural Network with memory
- The vanishing gradient problem in RNNs
- The LSTM model
- The GRU model
- *One/Many-to-one/many models*
- *Seq2Seq models*
- *Encoder and Decoder architectures*
- *Autoregressive RNNs*
- *(More on these during W8!)*

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