Restricted

50.039 Theory and Practice of Deep Learning

W5-S3 Times Series Data and Recurrent Neural Networks

Matthieu De Mari

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Restricted

Restricted

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?

Restricted

Restricted

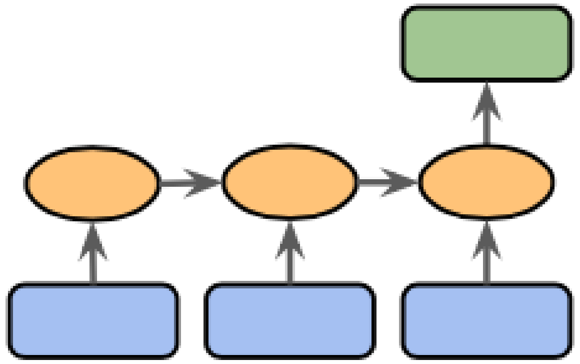
About this week (Week 5)

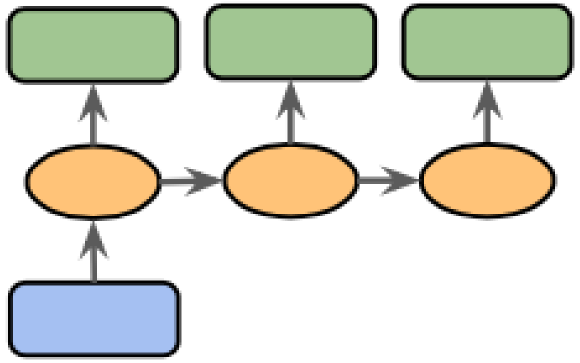
8. What is the **LSTM** model?
9. What is the **GRU** model?
10. What are **one-to-one**, **one-to-many**, **many-to-one** and **many-to-many** models?
11. What are some **typical application examples** of these?
12. What is a **Seq2Seq** model?
13. What are **encoder** and **decoder** architectures?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

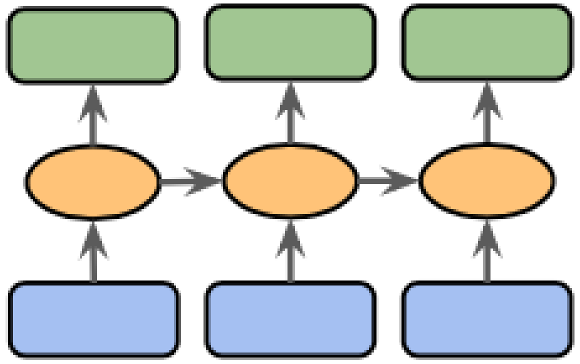
Restricted

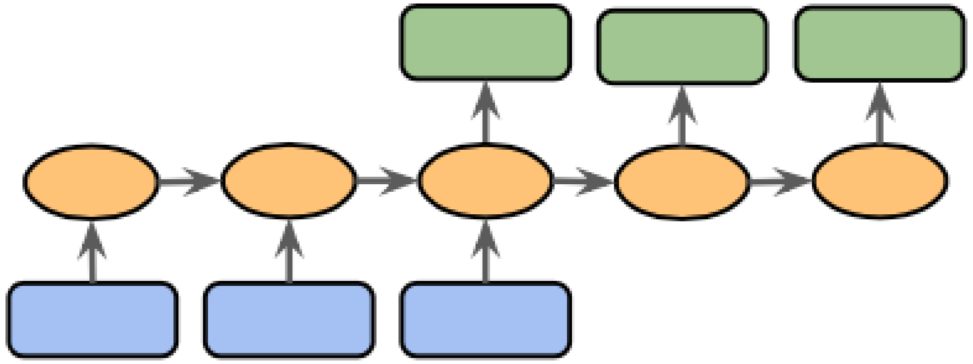
Restricted

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


many-to-one


one-to-many


many-to-many


many-to-many

Restricted

One-to-one models

Definition (one-to-one models):

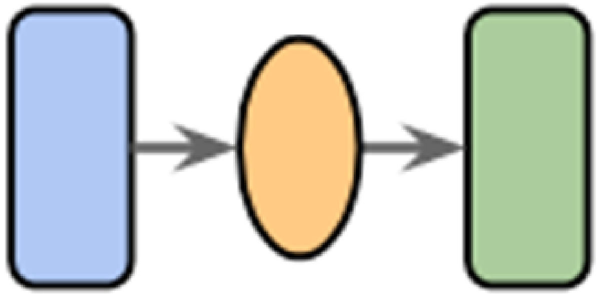
A **one-to-one** **model** is a machine learning model that:

- 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.



Restricted

Restricted

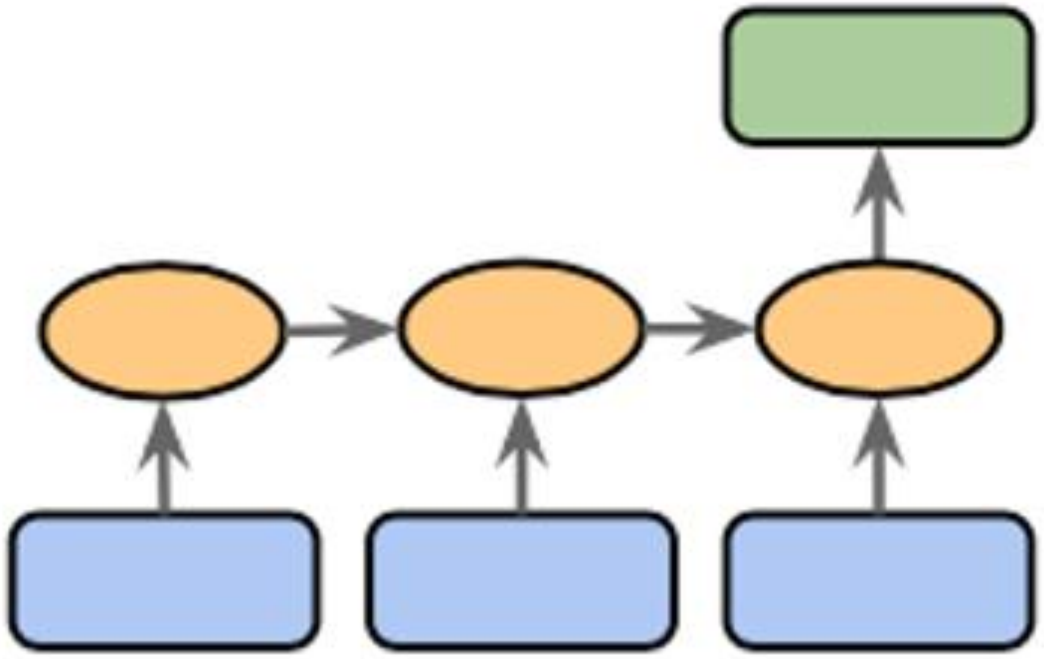
Many-to-one models

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.



Restricted

Restricted

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, already containing a few words **(sentence = sequence of words ≈ time series!).**

Restricted

Restricted

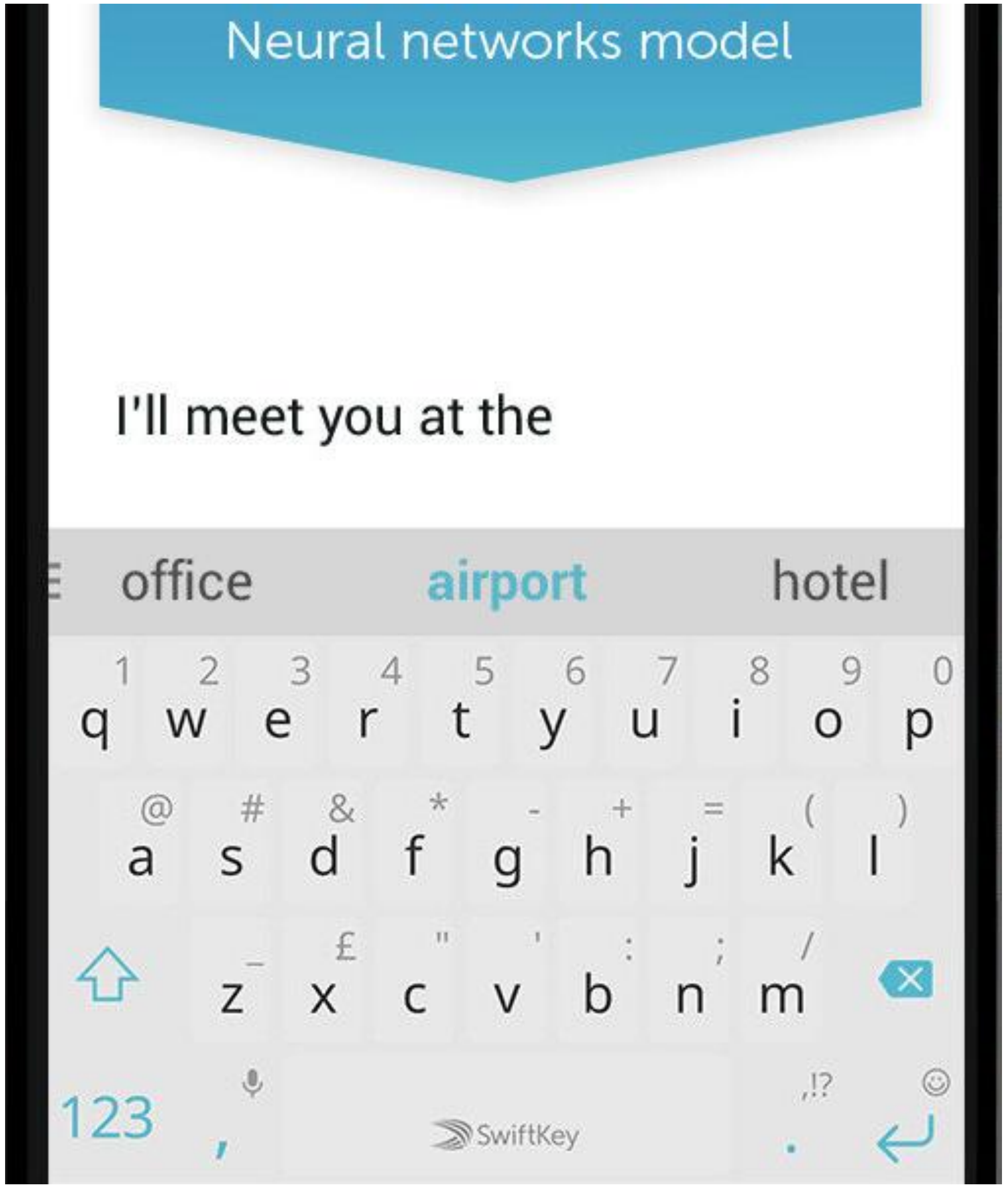
Many-to-one models

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.



Restricted

Restricted

Many-to-one models

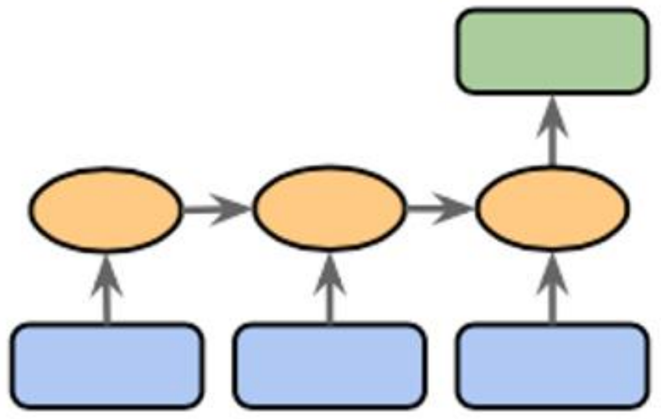
Definition (many-to-one models):

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.



Restricted

Restricted

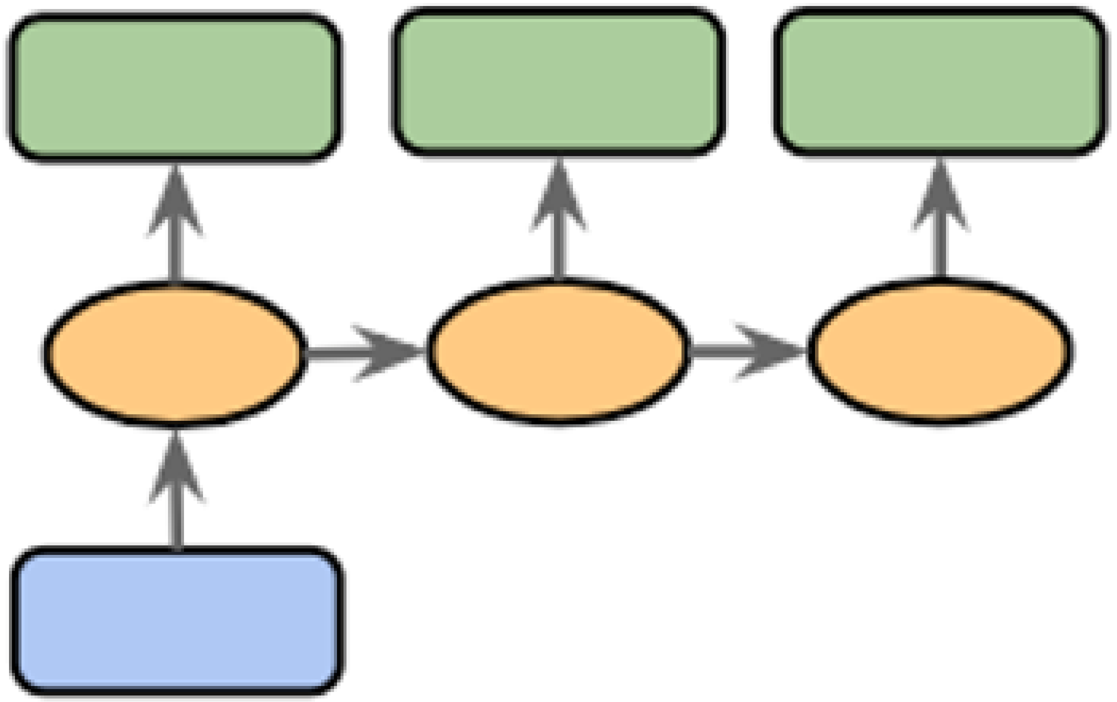
One-to-many models

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 problems, because most of the time we prefer to have a higher dimensionality on inputs rather than the outputs (Note that this might change later!)



Restricted

Restricted

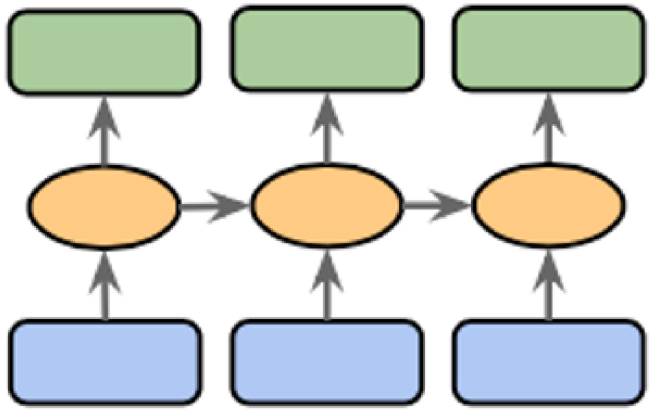
Many-to-many/Sequence-to-sequence models

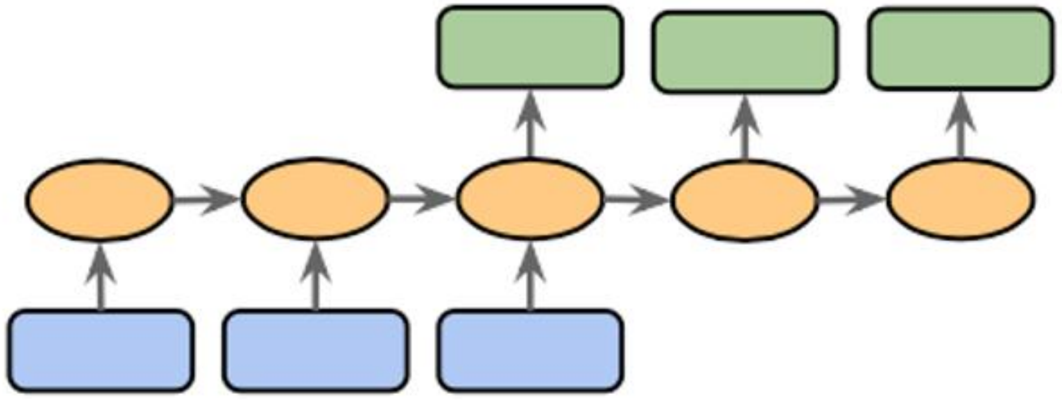
Definition (many-to-many models):

A **many-to-many 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 **<u>many</u> outputs**.

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





Restricted

Restricted

Many-to-many/Sequence-to-sequence 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.

Restricted

Restricted

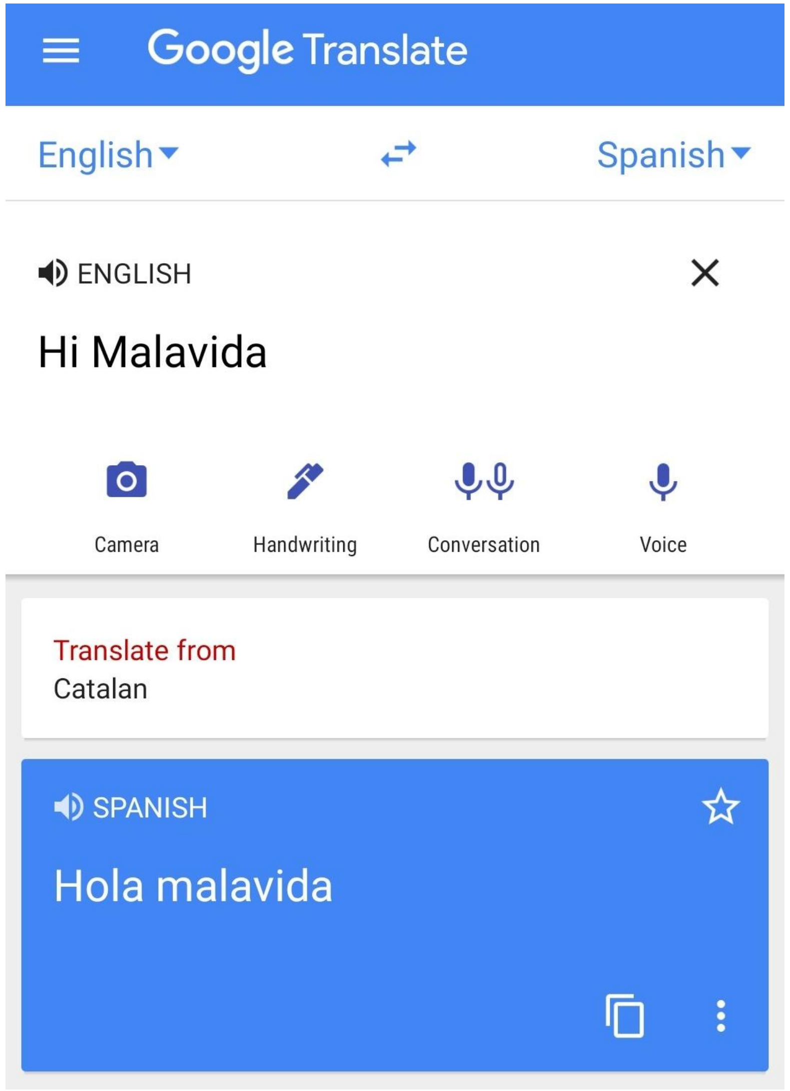
Many-to-many/Sequence-to-sequence models

Definition (many-to-many models):

A **many-to-many 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 **<u>many</u> outputs**.

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



Hi Malavida

Translate from Catalan

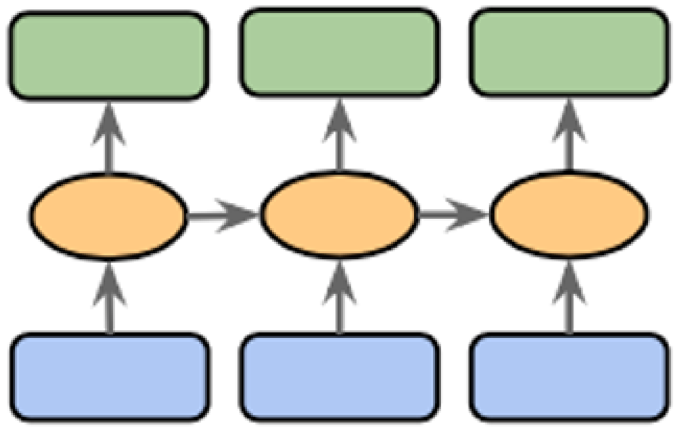
Hola malavida

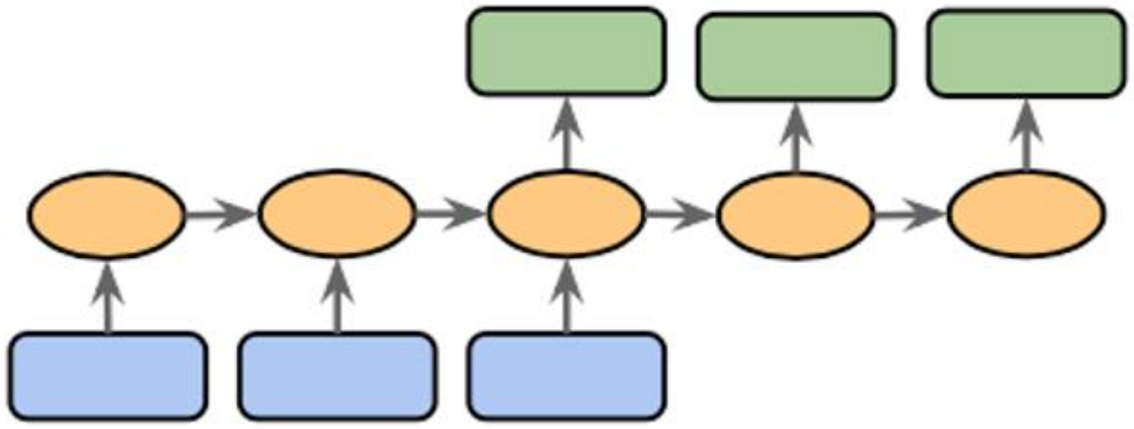
Restricted

Restricted

Many-to-many/Sequence-to-sequence models

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





Restricted

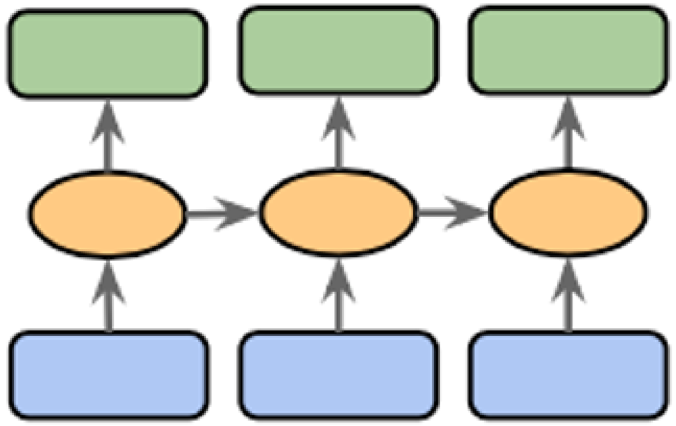
Restricted

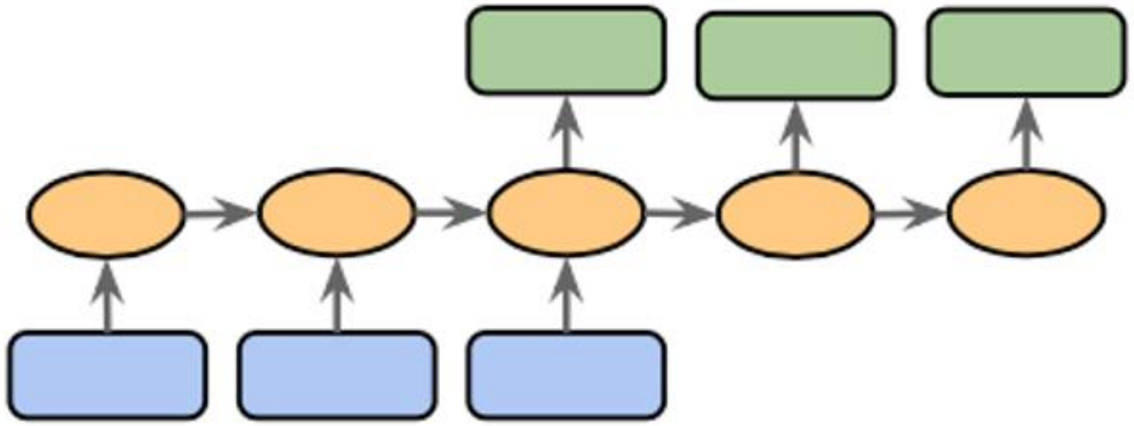
Encoder and Decoder models

Definition (Seq2Seq, encoder and decoder models):

A **Seq2Seq model** is a special type of many-to-many model. Its key difference relies in separating the analysis of the input sequence and the production of an output after having seen all inputs.

This is often handled by two differents part of a larger neural network, called an encoder and a decoder network.





Restricted

Restricted

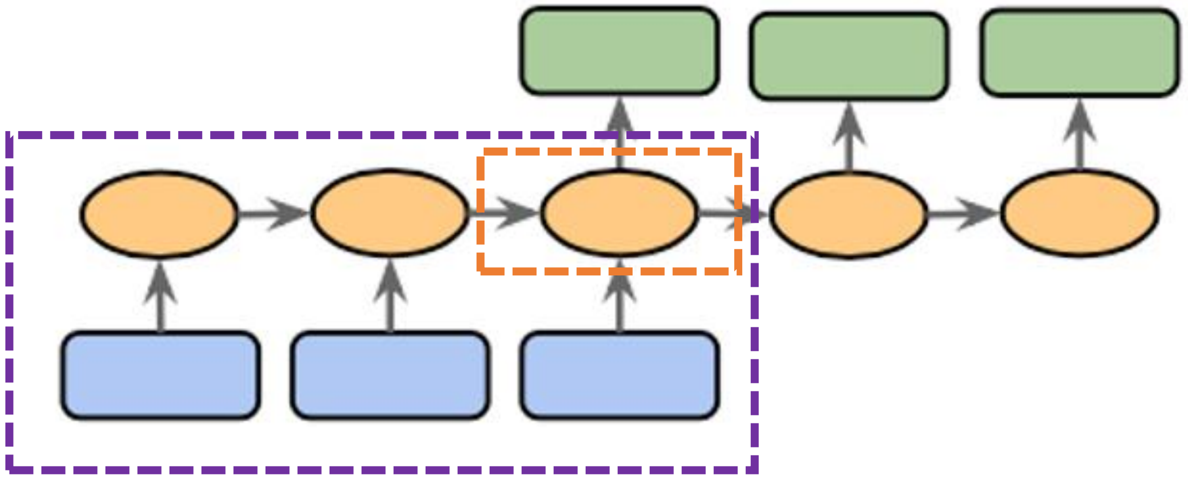
Encoder and Decoder models

Definition (Seq2Seq, 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.



Restricted

Restricted

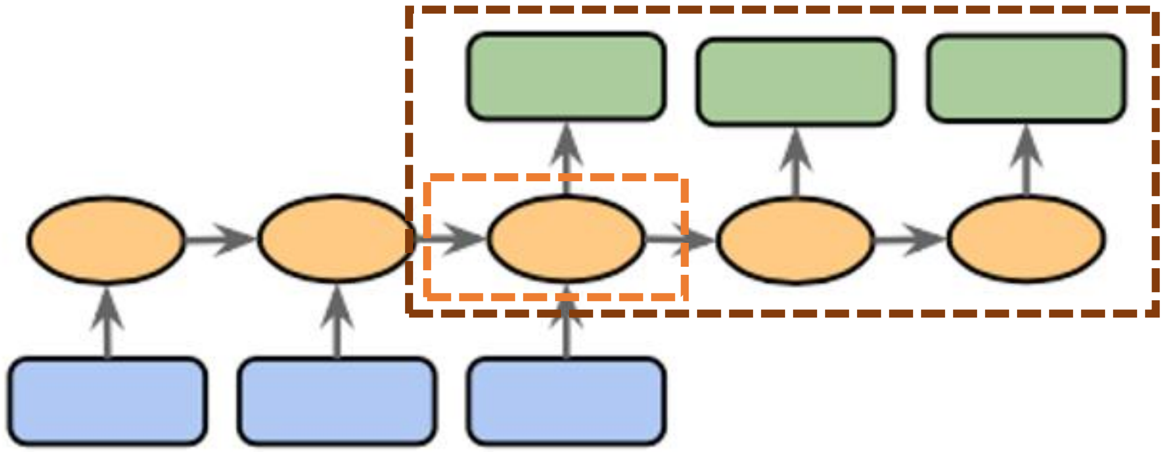
Encoder and Decoder models

Definition (Seq2Seq, 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.



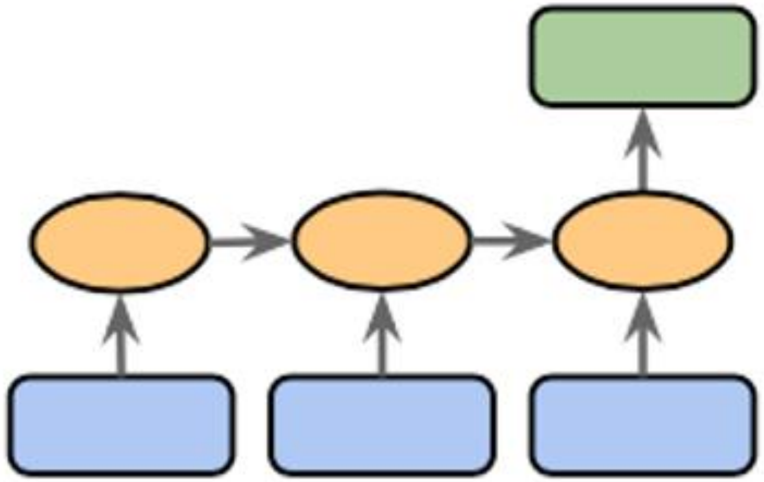
Restricted

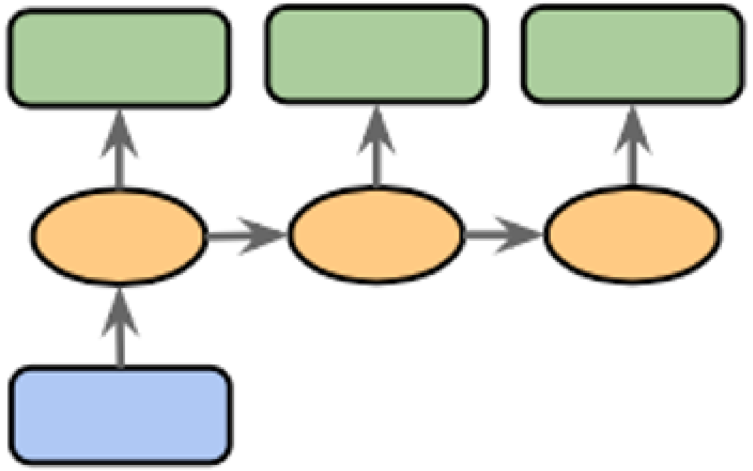
Restricted

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), that is receiving this single **encoding vector** as input and producing a sequence as output.



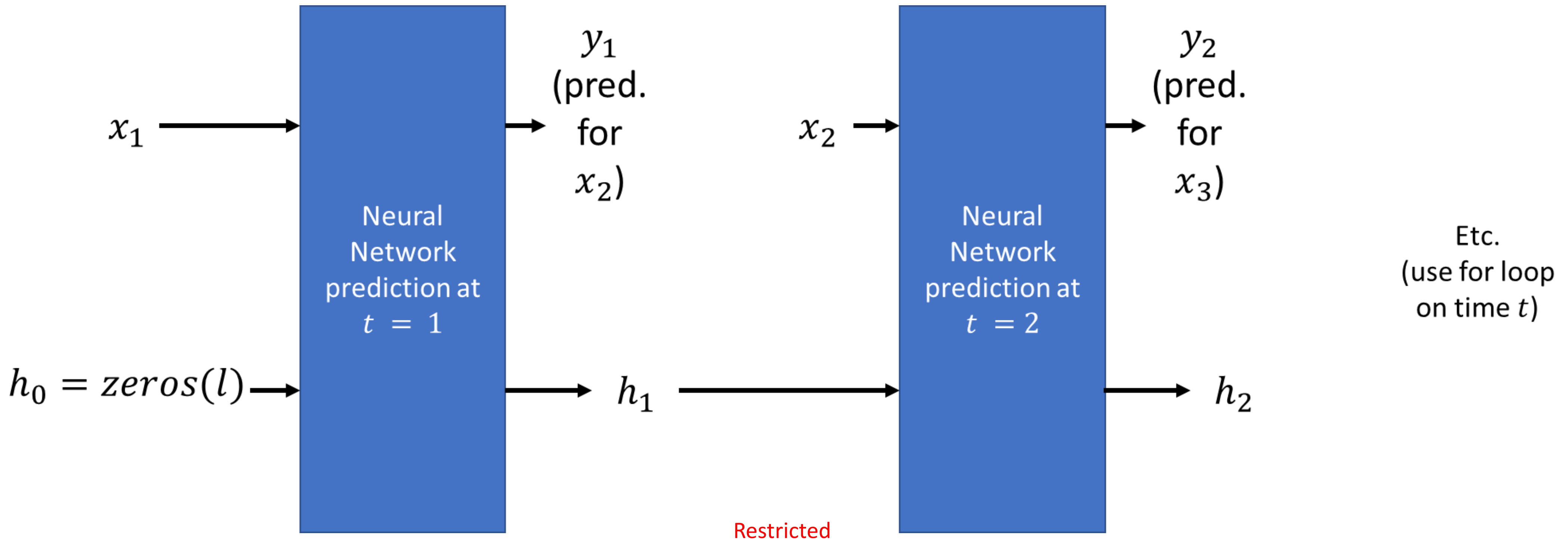


Restricted

Restricted

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 x_t** as input on each time step.

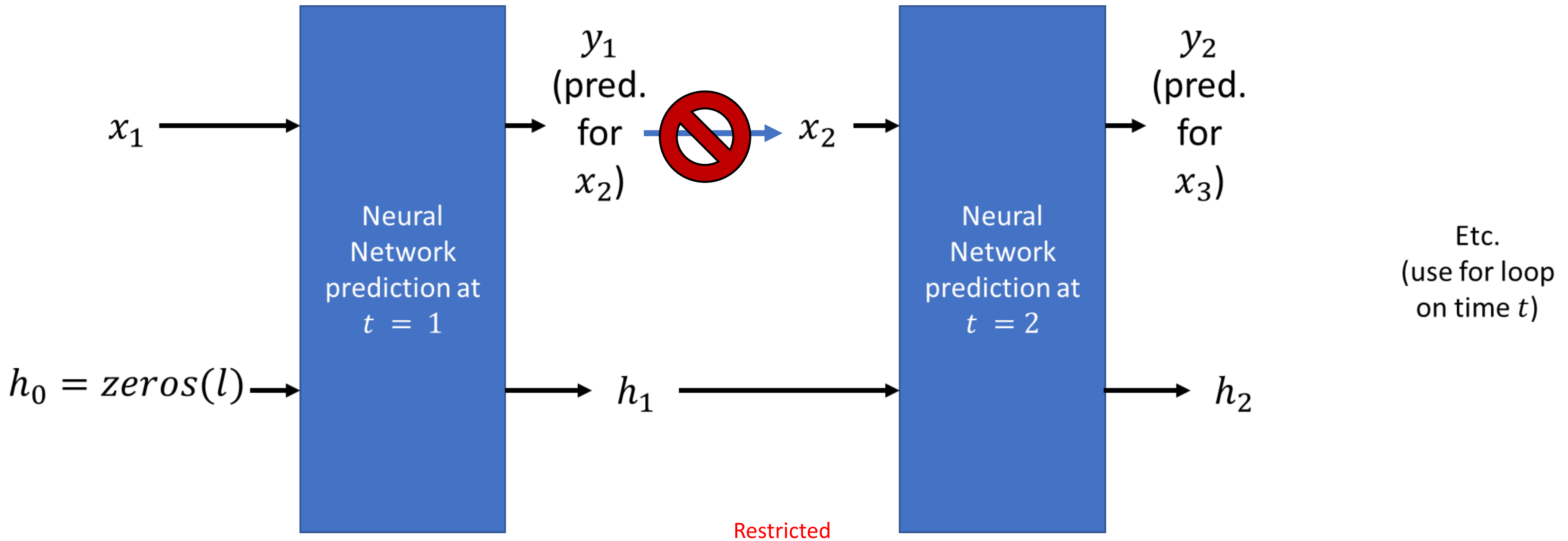

x_1
$h_0 = zeros(l)$
Neural Network prediction at $t = 1$
y_1 (pred. for x_2)
h_1
x_2
Neural Network prediction at $t = 2$
y_2 (pred. for x_3)
h_2
Etc. (use for loop on time t)

Restricted

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}!

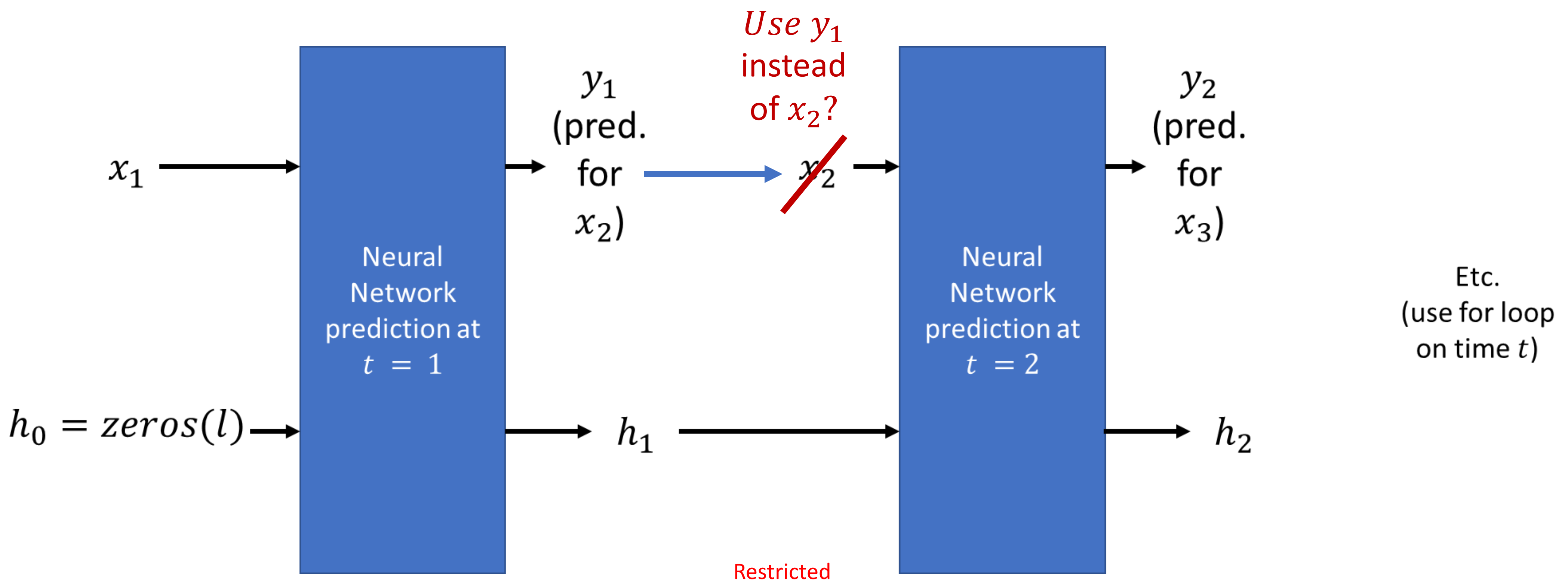

x_1
$h_0 = zeros(l)$
Neural Network prediction at $t = 1$
y_1 (pred. for x_2)
x_2
h_1
Neural Network prediction at $t = 2$
y_2 (pred. for x_3)
h_2
Etc. (use for loop on time t)

Restricted

Restricted

A quick word on autoregressive RNNs

But what if we did?


x_1
$h_0 = zeros(l)$
Neural Network prediction at $t = 1$
y_1 (pred. for x_2)
h_1
$Use\ y_1$ instead of x_2?
$\cancel{x_2}$
Neural Network prediction at $t = 2$
y_2 (pred. for x_3)
h_2
Etc. (use for loop on time t)

Restricted

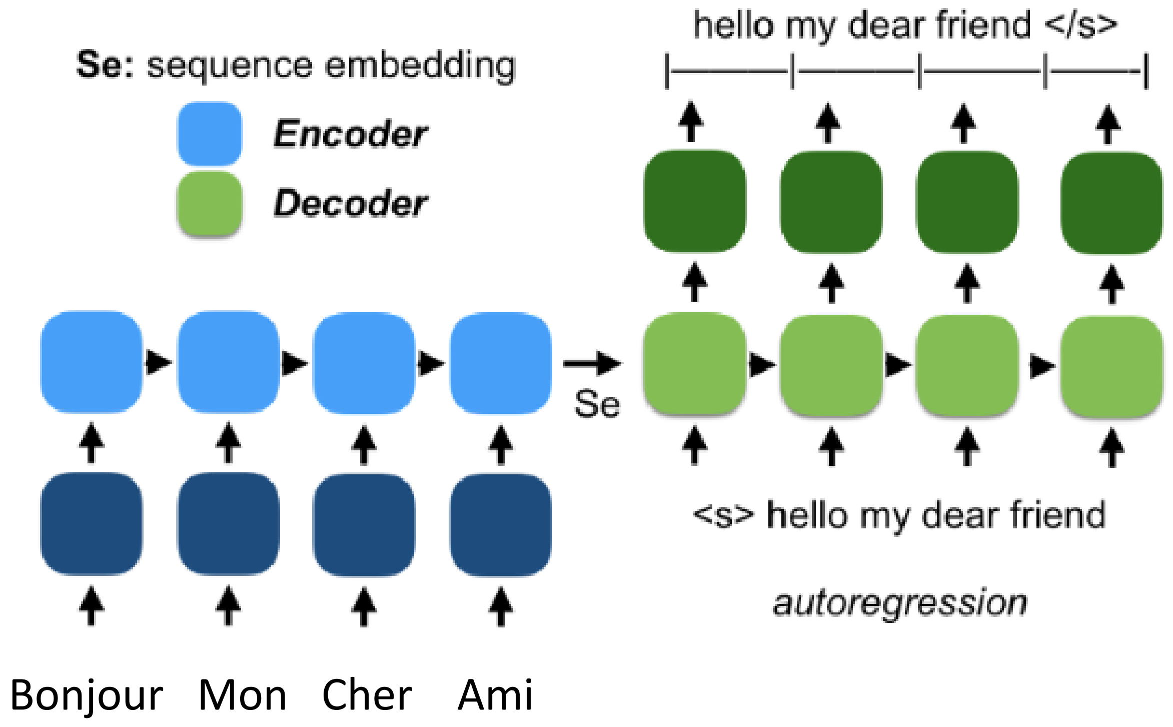
Restricted

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, you must use your own predictions y_t in place of x_{t+1}!



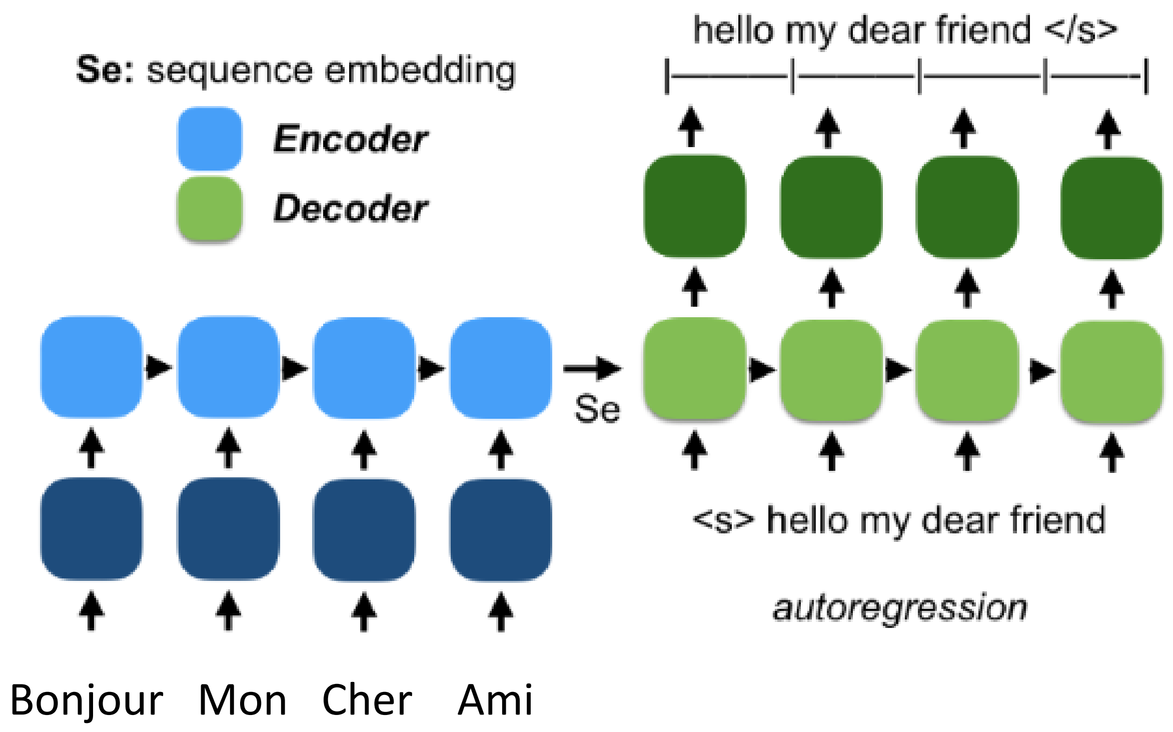
Restricted

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.

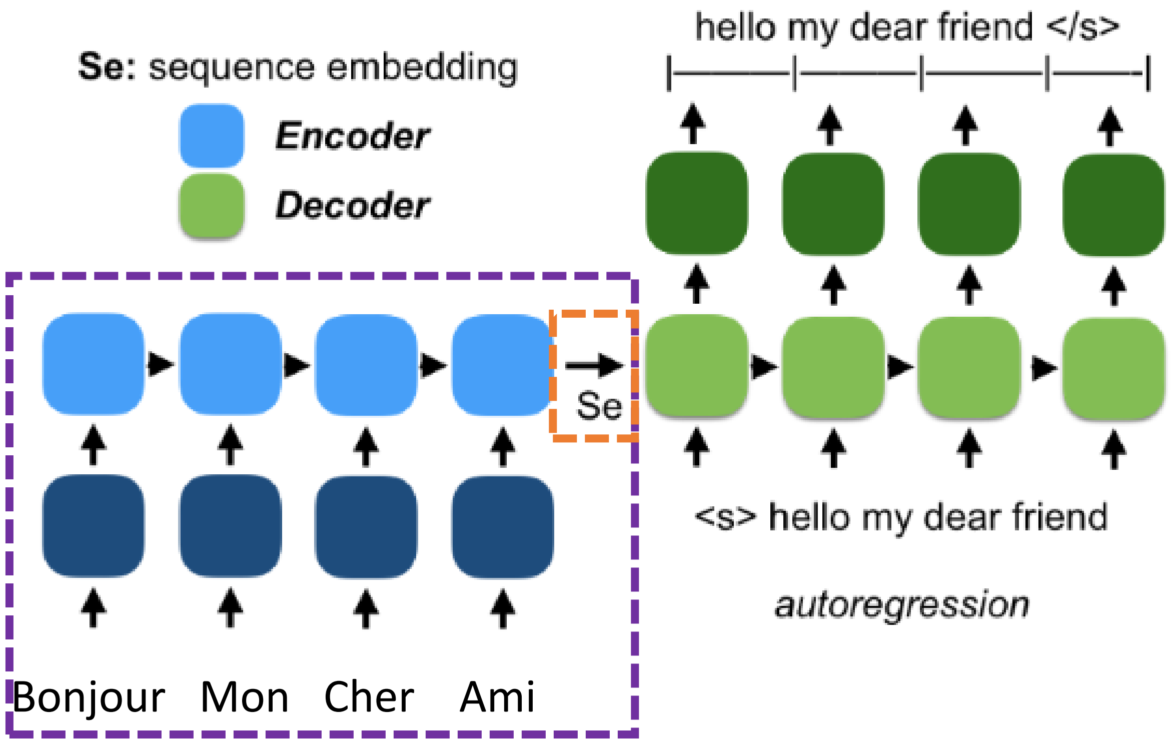


Restricted

Restricted

A quick word on autoregressive RNNs

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



Restricted

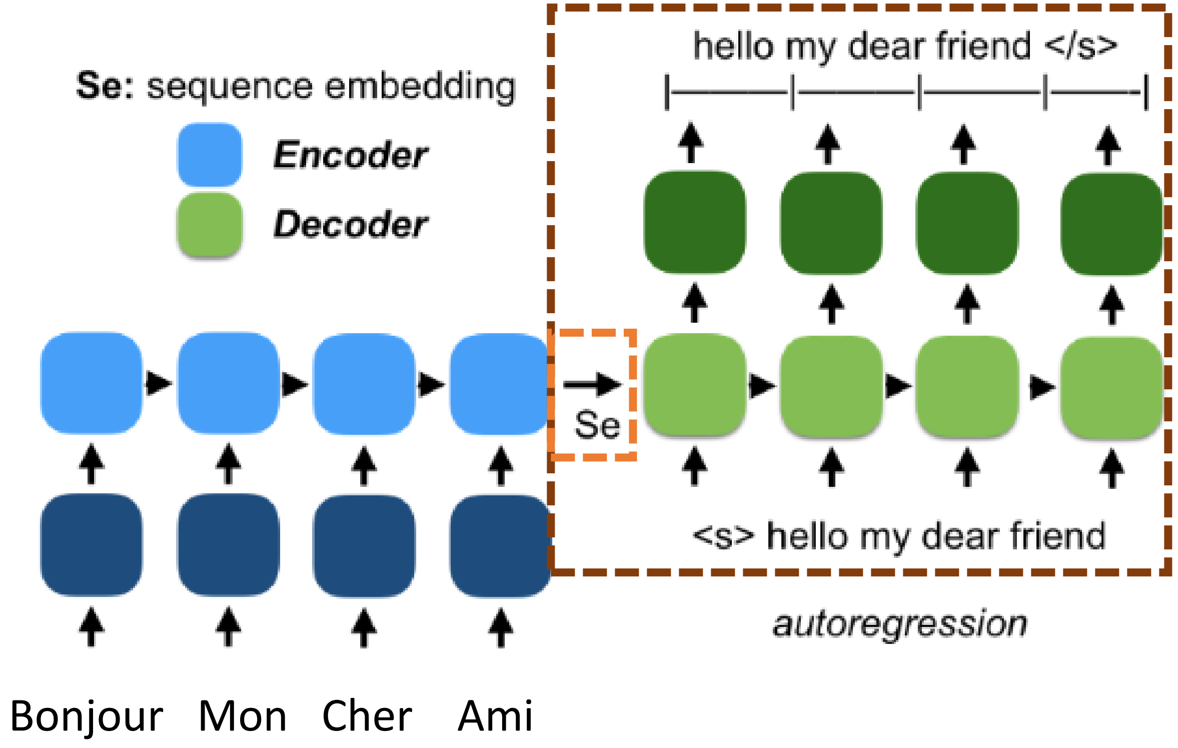
Restricted

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.



Restricted

Restricted

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 uses a decoder architecture of some sort to reconstructs an answer, a sentence, one word after the other!**

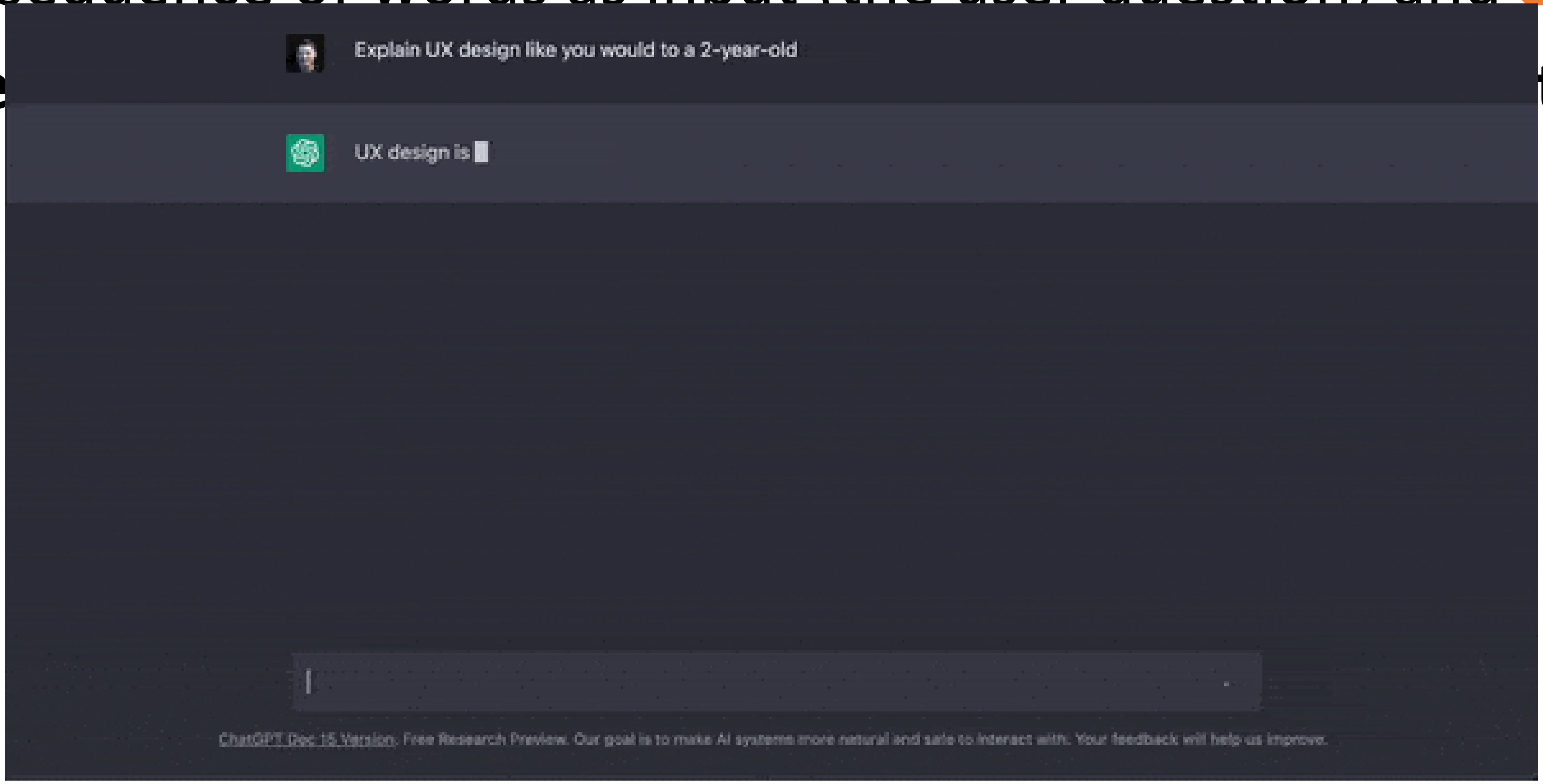
Restricted

Restricted

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 use ts an answer,**



Explain UX design like you would to a 2-year-old

UX design is

ChatGPT Dec 15 Version. Free Research Preview. Our goal is to make AI systems more natural and safe to interact with. Your feedback will help us improve.

Restricted

Restricted

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 uses a decoder architecture of some sort to reconstructs an answer, a sentence, one word after the other!**

We will learn more about these autoregressive models during Weeks 8 (Introduction to NLP) and 13 (ChatGPT)!

And during the NLP specialty course on Term 7!

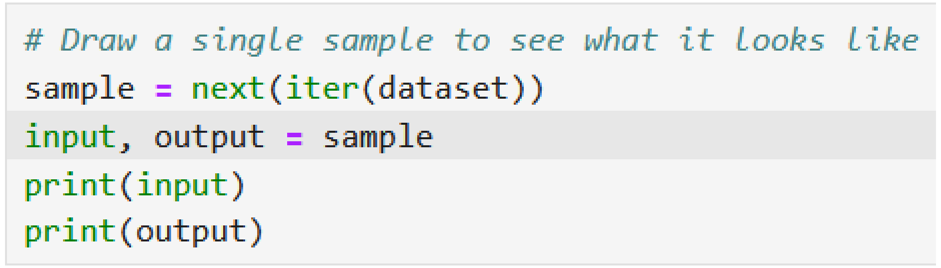
Restricted

Restricted

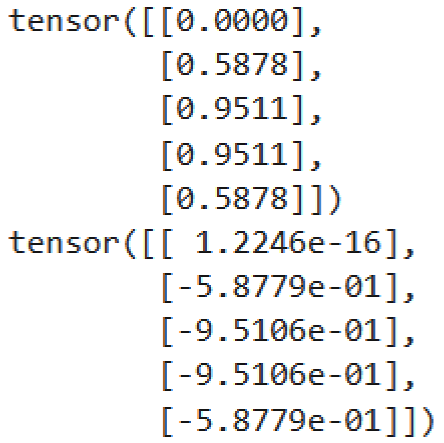
Implementation of a simple Seq2Seq

Just like before, we generate a synthetic dataset.

- Objective is predicting the next values of a sinsusoid curve.
- This time however, at each time t, the model will get to see the last five values of the curve as inputs $(x_{t-4}, x_{t-3}, x_{t-2}, x_{t-1}, x_t)$ and will attempt to predict the next five values $(x_{t+1}, x_{t+2}, x_{t+3}, x_{t+4}, x_{t+5})$.



```python
# Draw a single sample to see what it looks like
sample = next(iter(dataset))
input, output = sample
print(input)
print(output)
```



```
tensor([[0.0000],
        [0.5878],
        [0.9511],
        [0.9511],
        [0.5878]])
tensor([[ 1.2246e-16],
        [-5.8779e-01],
        [-9.5106e-01],
        [-9.5106e-01],
        [-5.8779e-01]])
```

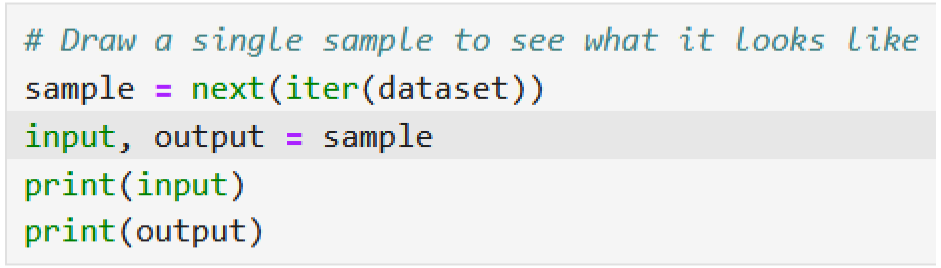
Restricted

Restricted

Implementation of a simple Seq2Seq

Just like before, we generate a synthetic dataset.

- Objective is predicting the next values of a sinsusoid curve.
- This time however, at each time t, the model will get to see the last five values of the curve as inputs $(x_{t-4}, x_{t-3}, x_{t-2}, x_{t-1}, x_t)$ and will attempt to predict the next five values $(x_{t+1}, x_{t+2}, x_{t+3}, x_{t+4}, x_{t+5})$.



```python
# Draw a single sample to see what it looks like
sample = next(iter(dataset))
input, output = sample
print(input)
print(output)
```

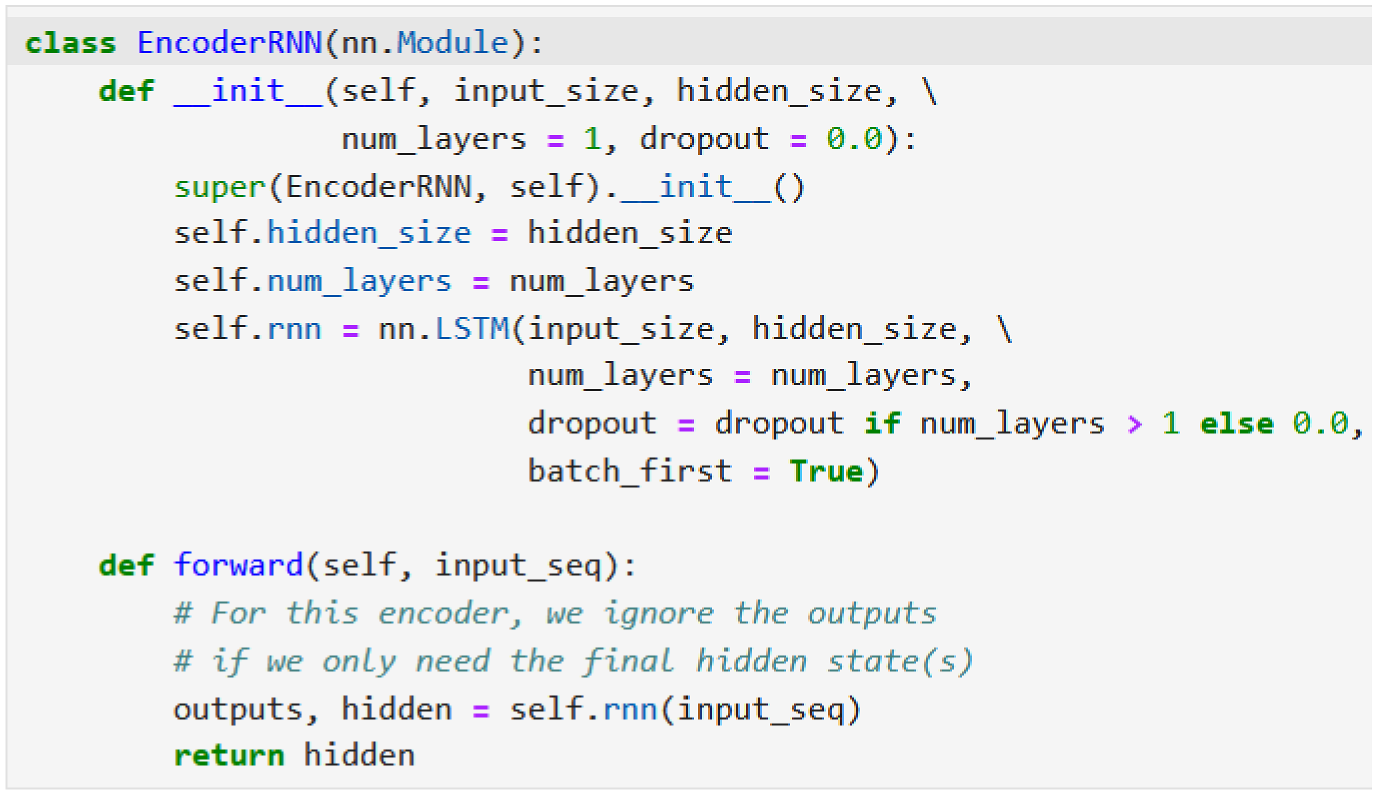
```
tensor([[0.0000],
        [0.5878],
        [0.9511],
        [0.9511],
        [0.5878]])
tensor([[ 1.2246e-16],
        [-5.8779e-01],
        [-9.5106e-01],
        [-9.5106e-01],
        [-5.8779e-01]])
```

Restricted

Restricted

Our Encoder RNN

- The EncoderRNN shown in Notebook 6 and 7 processes input sequences through several layers of LSTM.
- It uses several layers of LSTM and possibly dropout.
- It is designed to capture the temporal dependencies in the data and encode this input in an encoding vector.



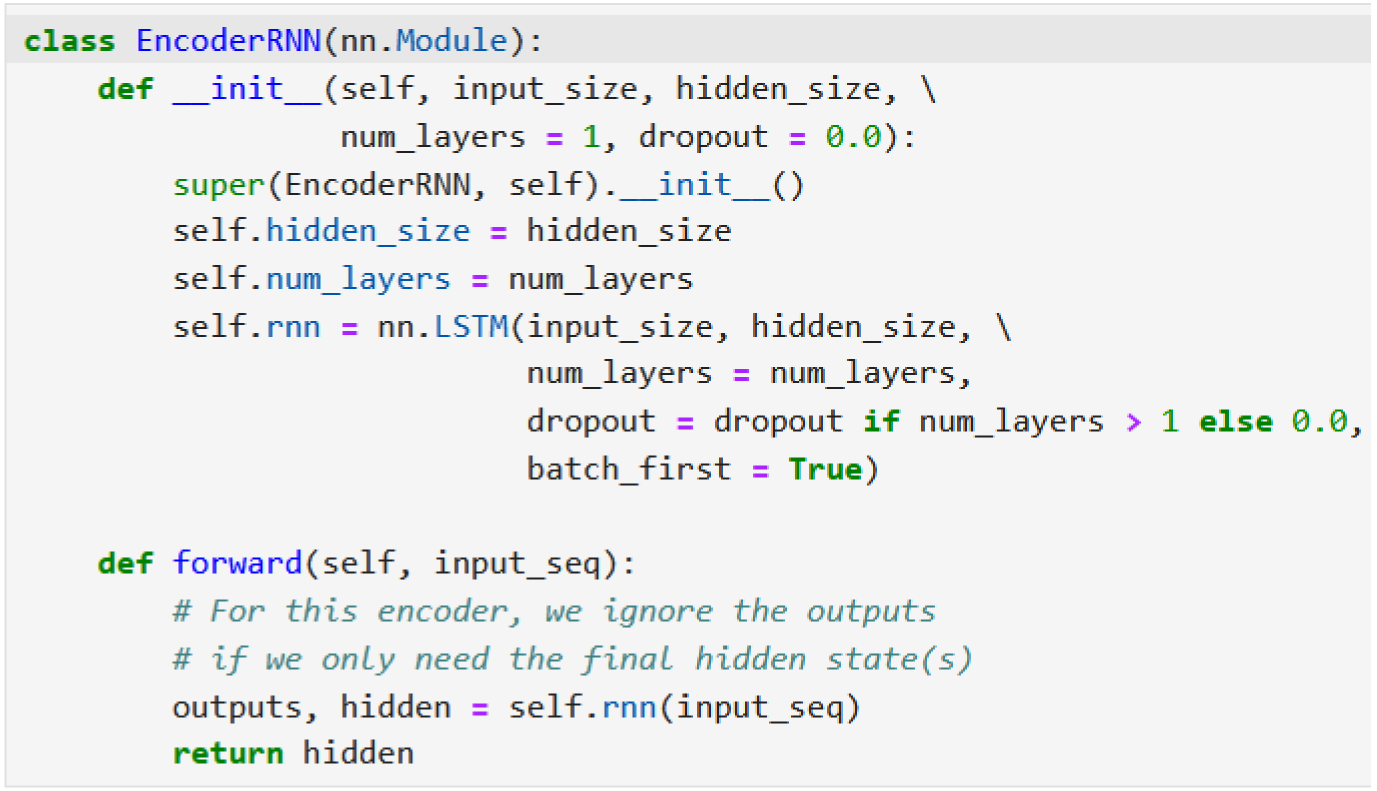
```
class EncoderRNN(nn.Module):
    def __init__(self, input_size, hidden_size, \
                 num_layers = 1, dropout = 0.0):
        super(EncoderRNN, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.rnn = nn.LSTM(input_size, hidden_size, \
                           num_layers = num_layers,
                           dropout = dropout if num_layers > 1 else 0.0,
                           batch_first = True)

    def forward(self, input_seq):
        # For this encoder, we ignore the outputs
        # if we only need the final hidden state(s)
        outputs, hidden = self.rnn(input_seq)
        return hidden
```

Restricted

Restricted

Our Encoder RNN

- Only the final hidden states are returned, encapsulating the learned representation of the entire sequence as an encoding vector.
- This encoder will later be used in a sequence-to-sequence model, where its final hidden state initializes the decoder to generate outputs based on the input sequence.



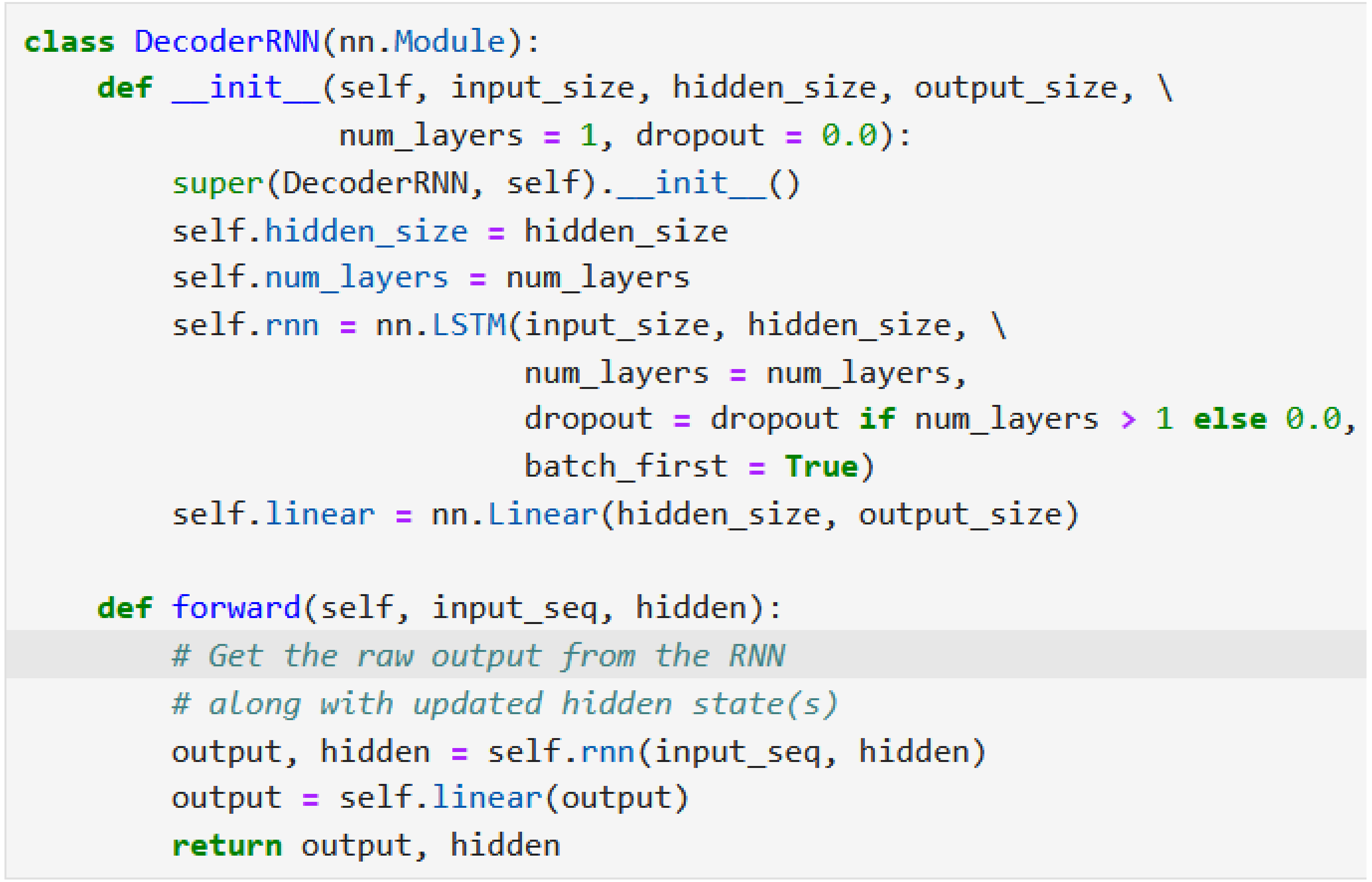
```
class EncoderRNN(nn.Module):
    def __init__(self, input_size, hidden_size, \
                 num_layers = 1, dropout = 0.0):
        super(EncoderRNN, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.rnn = nn.LSTM(input_size, hidden_size, \
                           num_layers = num_layers,
                           dropout = dropout if num_layers > 1 else 0.0,
                           batch_first = True)

    def forward(self, input_seq):
        # For this encoder, we ignore the outputs
        # if we only need the final hidden state(s)
        outputs, hidden = self.rnn(input_seq)
        return hidden
```

Restricted

Restricted

Our Decoder RNN

- The DecoderRNN class is designed to act as the decoder in a sequence-to-sequence architecture.
- It processes the encoding vector using an LSTM and then maps the LSTM outputs to the desired output space with a single and final linear layer.



```python
class DecoderRNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, \
                 num_layers = 1, dropout = 0.0):
        super(DecoderRNN, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.rnn = nn.LSTM(input_size, hidden_size, \
                           num_layers = num_layers,
                           dropout = dropout if num_layers > 1 else 0.0,
                           batch_first = True)
        self.linear = nn.Linear(hidden_size, output_size)

    def forward(self, input_seq, hidden):
        # Get the raw output from the RNN
        # along with updated hidden state(s)
        output, hidden = self.rnn(input_seq, hidden)
        output = self.linear(output)
        return output, hidden
```

Restricted

Restricted

Seq2Seq assembling Encoder and Decoder

Assembling our encoder and decoder classes from earlier, the Seq2Seq class implements a sequence-to-sequence model using an encoder-decoder architecture.

- The encoder processes the input sequence to produce a final hidden state, which is then used as the starting point for the memory vector of the decoder.
- The decoder is then responsible for generating the output sequence.

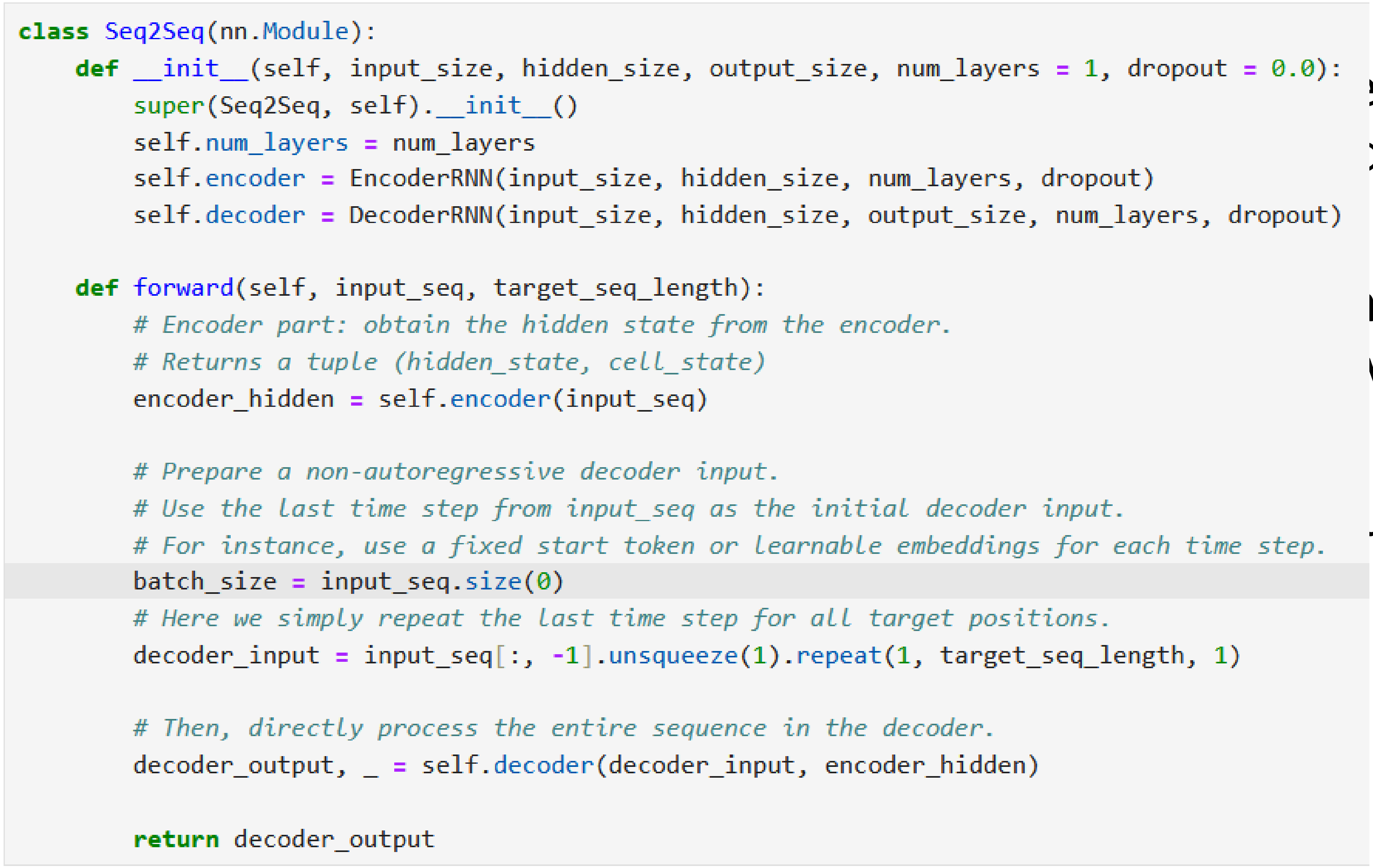
Restricted

Restricted

Seq2Seq assembling Encoder and Decoder

Assemb eq2Seq
class im der-
decode

- The e idden
  state, vector
  of the
- The d uence.



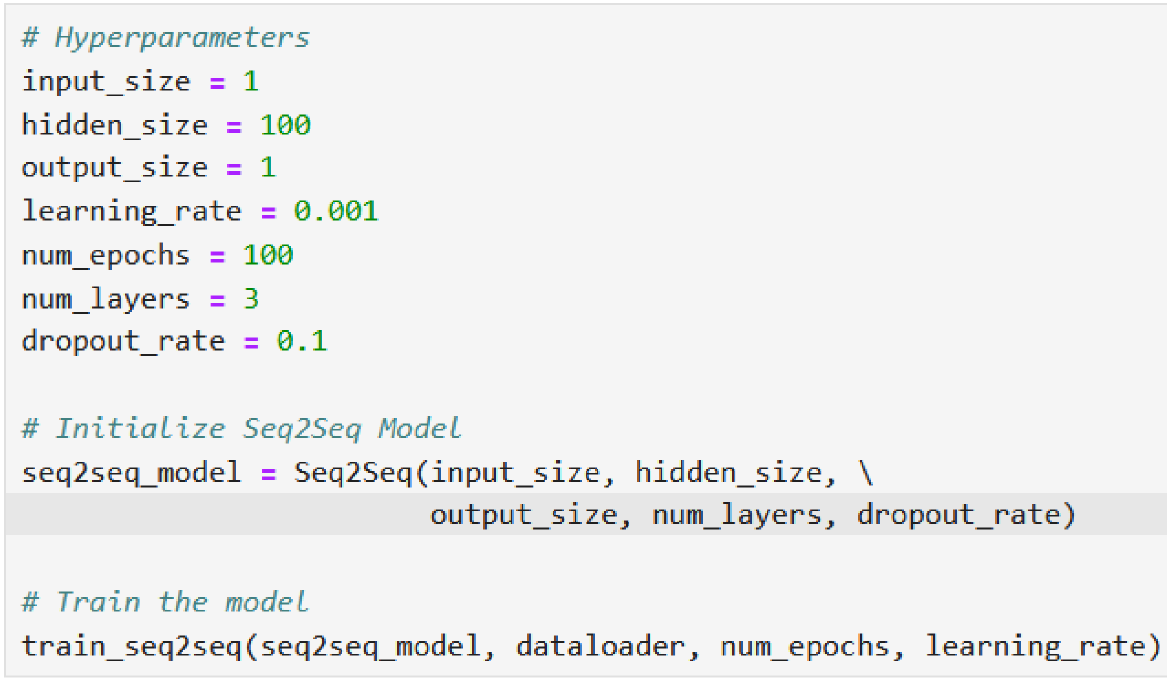
```
class Seq2Seq(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers = 1, dropout = 0.0):
        super(Seq2Seq, self).__init__()
        self.num_layers = num_layers
        self.encoder = EncoderRNN(input_size, hidden_size, num_layers, dropout)
        self.decoder = DecoderRNN(input_size, hidden_size, output_size, num_layers, dropout)

    def forward(self, input_seq, target_seq_length):
        # Encoder part: obtain the hidden state from the encoder.
        # Returns a tuple (hidden_state, cell_state)
        encoder_hidden = self.encoder(input_seq)

        # Prepare a non-autoregressive decoder input.
        # Use the last time step from input_seq as the initial decoder input.
        # For instance, use a fixed start token or learnable embeddings for each time step.
        batch_size = input_seq.size(0)
        # Here we simply repeat the last time step for all target positions.
        decoder_input = input_seq[:, -1].unsqueeze(1).repeat(1, target_seq_length, 1)

        # Then, directly process the entire sequence in the decoder.
        decoder_output, _ = self.decoder(decoder_input, encoder_hidden)

        return decoder_output
```

Restricted

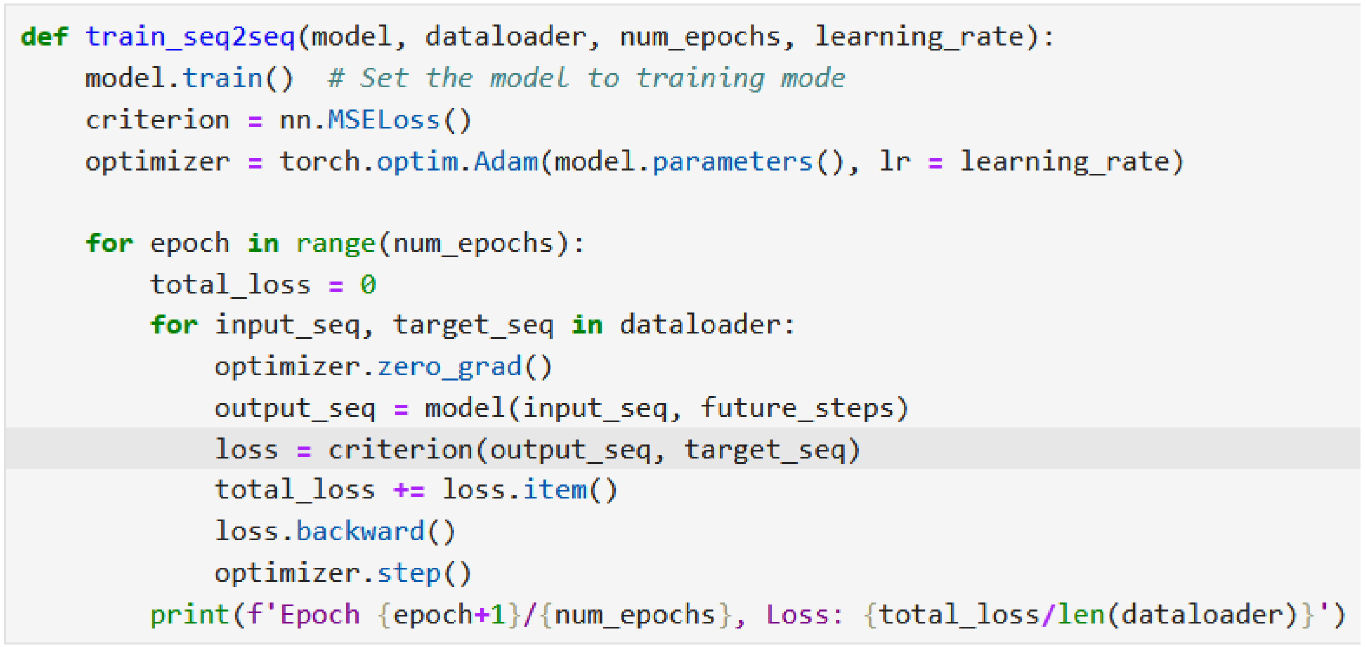
Restricted

Trainer is kept simple

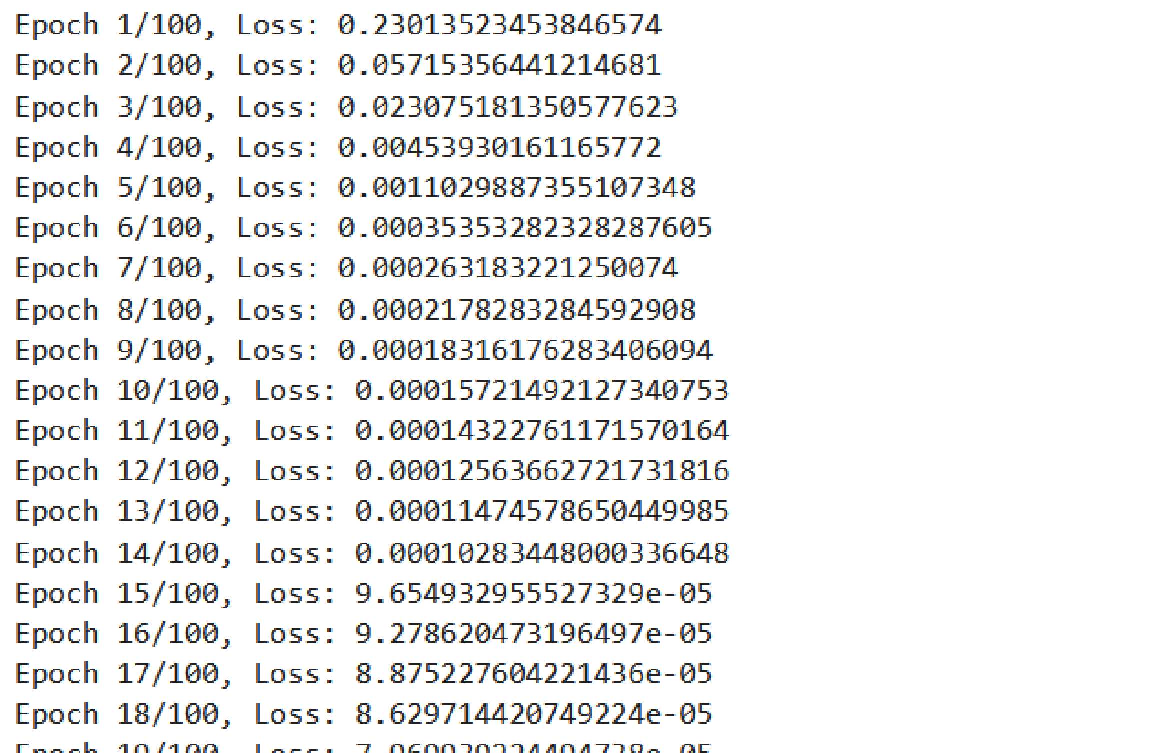
- And it trains nicely!



```python
# Hyperparameters
input_size = 1
hidden_size = 100
output_size = 1
learning_rate = 0.001
num_epochs = 100
num_layers = 3
dropout_rate = 0.1

# Initialize Seq2Seq Model
seq2seq_model = Seq2Seq(input_size, hidden_size, \
                        output_size, num_layers, dropout_rate)

# Train the model
train_seq2seq(seq2seq_model, dataloader, num_epochs, learning_rate)
```



```python
def train_seq2seq(model, dataloader, num_epochs, learning_rate):
    model.train()  # Set the model to training mode
    criterion = nn.MSELoss()
    optimizer = torch.optim.Adam(model.parameters(), lr = learning_rate)

    for epoch in range(num_epochs):
        total_loss = 0
        for input_seq, target_seq in dataloader:
            optimizer.zero_grad()
            output_seq = model(input_seq, future_steps)
            loss = criterion(output_seq, target_seq)
            total_loss += loss.item()
            loss.backward()
            optimizer.step()
        print(f'Epoch {epoch+1}/{num_epochs}, Loss: {total_loss/len(dataloader)}')
```



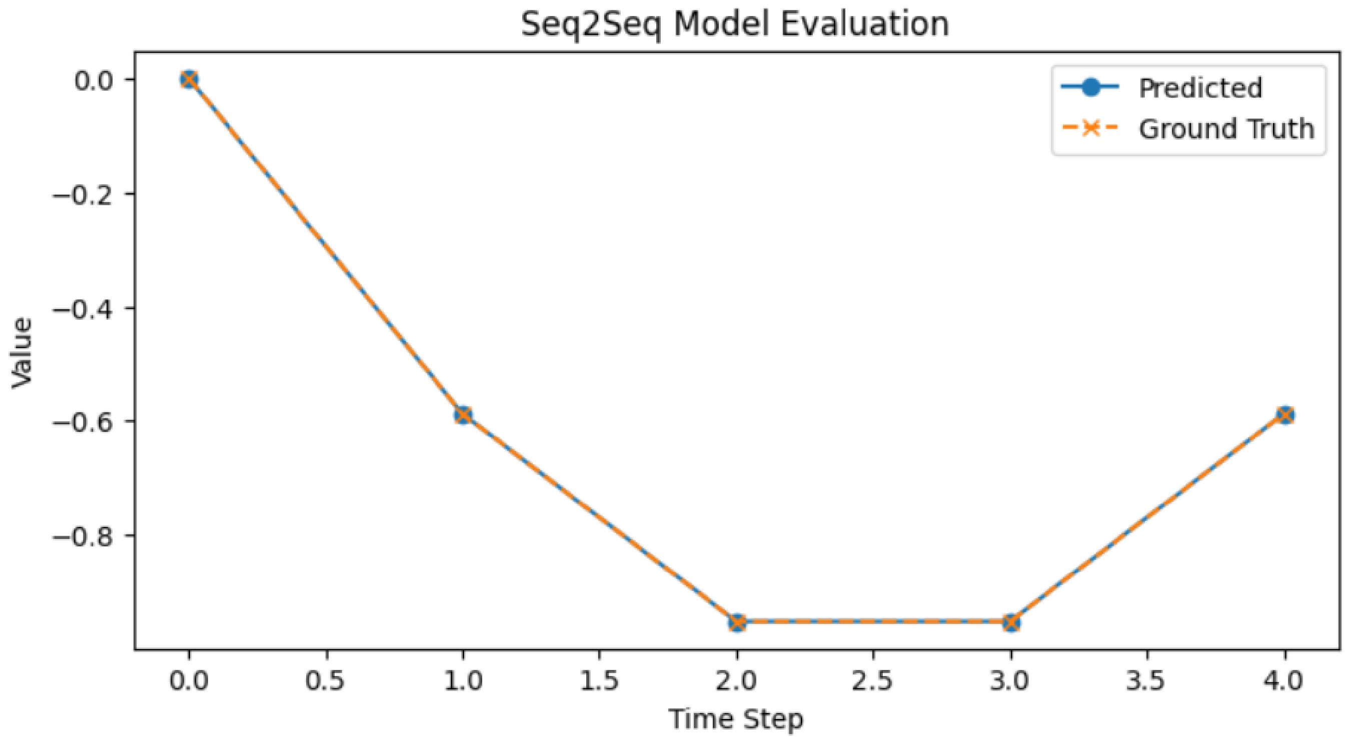
```
Epoch 1/100, Loss: 0.23013523453846574
Epoch 2/100, Loss: 0.05715356441214681
Epoch 3/100, Loss: 0.023075181350577623
Epoch 4/100, Loss: 0.0045393016165772
Epoch 5/100, Loss: 0.0011029887355107348
Epoch 6/100, Loss: 0.00035353282328287605
Epoch 7/100, Loss: 0.000263183221250074
Epoch 8/100, Loss: 0.0002178283284592908
Epoch 9/100, Loss: 0.00018316176283406094
Epoch 10/100, Loss: 0.00015721492127340753
Epoch 11/100, Loss: 0.00014322761171570164
Epoch 12/100, Loss: 0.00012563662721731816
Epoch 13/100, Loss: 0.00011474578650449985
Epoch 14/100, Loss: 0.00010283448000336648
Epoch 15/100, Loss: 9.654932955527329e-05
Epoch 16/100, Loss: 9.278620473196497e-05
Epoch 17/100, Loss: 8.875227604221436e-05
Epoch 18/100, Loss: 8.629714420749224e-05
Epoch 19/100, Loss: [illegible]
```

Restricted

Restricted

Trainer is kept simple

- And it trains nicely!
- Confirm training by sampling and testing predictions!



Restricted

Restricted

Making it autoregressive

In this first implementation of a Seq2Seq (Notebook 6), we tried to generate the whole output sequence in one go.

- **Start:** The decoder input is prepared by taking the last time step of the encoder's input and then simply repeating it for every time step you want to predict.
- **Parallel Processing:** The decoder then processes this entire repeated input in one pass, generating all the outputs at once.
- **Result:** All parts of the output sequence are generated simultaneously, without depending on previous outputs.

Restricted

Restricted

Making it autoregressive

If we want to make it an **autoregressive RNN** (which is shown in Notebook 7), a few changes need to occur:

- **Start:** The decoder begins with a starting input (taken from the last time step of the encoder’s input), as before.
- **Loop:** For each time step, the decoder produces a single output of the sequence. Using a loop, that output is then fed back as the input to be used by the decoder to generate the next output of the sequence for the next time step.
- **Result:** The final output sequence is built one element at a time, with each prediction depending on the one before it.

Restricted

Restricted

Making it autoregressive

If we want to make it an **autoregressive RNN** (which is shown in Notebook 7), a few changes need to occur:

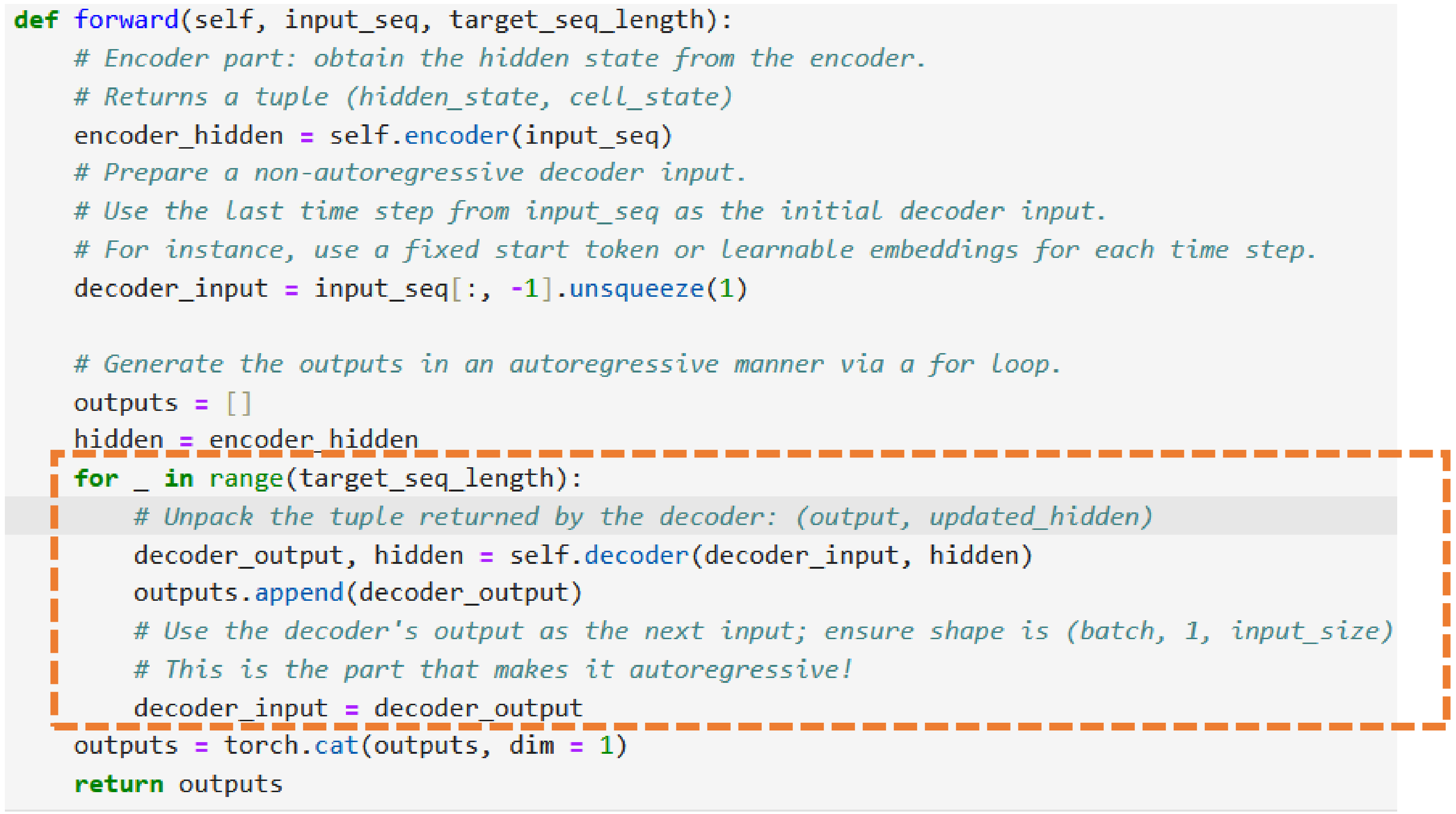
- **Start:** The decoder begins with a starting input (taken from the last time step of the encoder's input), as before.
- **Loop:** For each time step, the decoder produces a single output of the sequence. Using a loop, that output is then fed back as the input to be used by the decoder to generate the next output of the sequence for the next time step.
- **Result:** The final output sequence is built one element at a time, with each prediction depending on the one before it.

Restricted

Restricted

Making it autoregressive

This requires a minimal adjustment in the forward method of the Seq2Seq model to make the magic happen.



```python
def forward(self, input_seq, target_seq_length):
    # Encoder part: obtain the hidden state from the encoder.
    # Returns a tuple (hidden_state, cell_state)
    encoder_hidden = self.encoder(input_seq)
    # Prepare a non-autoregressive decoder input.
    # Use the last time step from input_seq as the initial decoder input.
    # For instance, use a fixed start token or learnable embeddings for each time step.
    decoder_input = input_seq[:, -1].unsqueeze(1)

    # Generate the outputs in an autoregressive manner via a for loop.
    outputs = []
    hidden = encoder_hidden
    for _ in range(target_seq_length):
        # Unpack the tuple returned by the decoder: (output, updated_hidden)
        decoder_output, hidden = self.decoder(decoder_input, hidden)
        outputs.append(decoder_output)
        # Use the decoder's output as the next input; ensure shape is (batch, 1, input_size)
        # This is the part that makes it autoregressive!
        decoder_input = decoder_output
    outputs = torch.cat(outputs, dim = 1)
    return outputs
```

Restricted

Go autoregressive or not?

Why Choose an Autoregressive Model?

- **Sequential Dependency:** Autoregressive models generate output one step at a time, with each new prediction influenced by the previous ones. This is especially beneficial when the sequence elements have strong dependencies—such as in natural language (Week 8!), where each word affects the next.
- **High Accuracy in Complex Tasks:** Because each output can be conditioned on the previously generated outputs, autoregressive models tend to produce more coherent and contextually appropriate results. This is why they're often used in tasks like machine translation, language modeling, and text generation.
- **Flexibility with Variable-Length Output:** Since the model generates output sequentially, it can easily adapt to generating sequences of different lengths, stopping when a special end-of-sequence token is produced.

Restricted

Restricted

Go autoregressive or not?

Why not use an autoregressive model?

- **Reduced Inference Time and Efficient Batch Prediction:**
  In time series forecasting, non-autoregressive models predict future values directly from a fixed window of observed historical data. This contrasts with autoregressive models, which recursively feed each prediction back into the model to generate the next step. By removing the recursive dependency on previous predictions, non-autoregressive models can compute all future values in parallel (at least during training and one-step-ahead prediction), or efficiently process entire sequences in batches, leading to faster inference. This makes non-autoregressive approaches particularly attractive for batch forecasting in operational systems where large numbers of time series need to be processed simultaneously.

Restricted

Restricted

Go autoregressive or not?

Why not use an autoregressive model?

- **Simpler Training for Certain Tasks:** For tasks where the outputs do not depend heavily on each other or where dependencies can be modelled in another way, non-autoregressive approaches may simply work better than autoregressive ones.

Restricted

Restricted

Go autoregressive or not?

In Summary…

- **Autoregressive models** are preferred when capturing the sequential nature of data is crucial, providing higher accuracy and more coherent outputs at the cost of slower, step-by-step generation. They are usually harder to train but exhibit better performance.
- **Non-autoregressive models** are ideal when speed and parallel processing are critical, even though they might sacrifice some accuracy or contextual coherence since each output is generated independently.

As in all things until now, try and see what works best for your project!

Restricted

Restricted

Conclusion (Week 5)

- 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!)

Restricted

Restricted

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.

Restricted

Restricted

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/

Restricted

Restricted

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/

Restricted

Restricted

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

Restricted