Gated Recurrent Unit
Long Short Term Memory
Sequence to Sequence

CITS4012 Natural Language Processing

A/Prof. Wei Liu

wei.liu@uwa.edu.au

Computer Science and Software Engineering
The University of Western Australia

May 12, 2022

THE UNIVERSITY OF WESTERN AUSTRALIA

What we are going to cover today

1. Gated Recurrent Units (GRU)
   - Reset and Update Gates
   - GRU Cell
   - GRU Layer
2. Long Short Term Memory (LSTM)
   - Hidden State and Cell State
   - LSTM in PyTorch
3. Sequence to Sequence
   - Encoder-Decoder Architecture
   - Sequence Prediction Example

[see Deep Learning with Pytorch Step by Step, Chapter 8 & 9]
[see NLP with PyTorch, Chapter 7 & 8]

THE UNIVERSITY OF WESTERN AUSTRALIA

Issues with Elman RNN

The Elman RNN cell adds the transformed hidden state t_h with t_x without any weighting:

- what if the data point adds more information than the previous hidden state?

It also discard (forget completely) the previous hidden state, and only use the newly computed one for the next cell iteration.

- what if the previous hidden state contains more information than the newly computed one?

There is no way for Elman RNN to address the two questions above.

Gated Recurrent Units, or GRUs for short, are the answer to those two questions!

Gated Recurrent Units (GRU)

Weight the Transformed Hidden State

Instead of computing the new hidden state by simply adding up t_h and t_x, we try scaling t_h first:

$$h_{new} = tanh(r * t_h + t_x)$$

The new parameter r controls how much we keep from the old hidden state before adding the transformed input.

- **low values of r**, the **relative importance** of **the input data point** is **increased**.
- We can recover the Elman RNN by **setting r to one**.

Remember Some of the Old Hidden State

Instead of simply computing a new hidden state and going with it, GRUs try a weighted average of both hidden states, old and new:

$$h_{new} = tanh(t_h + t_x)$$
$$h^{'} = h_{new} * (1 - z) + h_{old} * z$$

The new parameter z controls how much weight it should give to the old hidden state.

z controls how much we should remember from the old hidden state. Setting it zero, will get us back the Elman RNN!

Put it all together

Both r and z must *produce values between zero and one* thus allowing only a fraction of the original values to go through.

Gates

- The new parameters r is called the **reset gate**.
- The new parameter z is called the **update gate**.

$$h^{'} = tanh(r * t_h + t_x) * (1 - z) + h * z$$

- Every gate produces a vector of values (each value between zero and one) with a size corresponding to the number of hidden dimensions. For two hidden dimensions, a gate may have values like [0.52, 0.87] for example.
- Since gates produce vectors, operations involving them are element-wise multiplications ($*$).

THE UNIVERSITY OF WESTERN AUSTRALIA

Learning the Gates

Elman RNN is a special case of GRU ($r = 1$ and $z = 0$)

$$\text{RNN:} h^{'} = tanh(t_h + t_x)$$

$$\text{GRU:} h^{'} = \underbrace{\underbrace{tanh(r * t_{hn} + t_{xn})}_{\text{n}} * (1 - z) + h * z}_{\text{weighted average of n}}$$

Where do r and z come from?

Both gates are trained using a structure that is pretty much an RNN cell, except that it uses a sigmoid activation function to get values between $[0, 1]$.

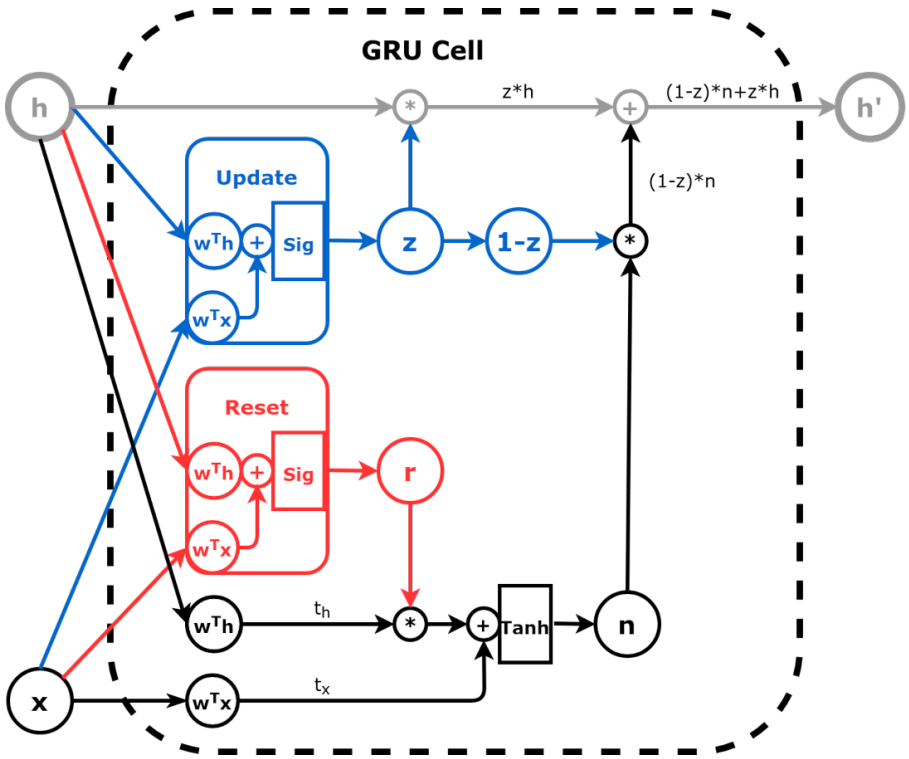
$$r = \sigma(t_{hr} + t_{xh})$$

$$z = \sigma(t_{hz} + t_{xz})$$

$$n = tanh(r * t_{hn} + t_{xn})$$

GRU Cell

$$
\begin{aligned}
r\ (hidden):\ t_{hr} &= W_{hr}\ \ h+\ \ b_{hr} \\
r\ (input):\ t_{xr} &= W_{ir}\ \ x+\ \ b_{ir} \\
z\ (hidden):\ t_{hz} &= W_{hz}\ \ h+\ \ b_{hz} \\
z\ (input):\ t_{xz} &= W_{iz}\ \ x+\ \ b_{iz} \\
n\ (hidden):\ t_{hn} &= W_{hn}\ \ h+\ \ b_{hn} \\
n\ (input):\ t_{xn} &= W_{in}\ \ x+\ \ b_{in}
\end{aligned}
$$



- red - rest gate (r); blue - update gate (z);
- black - (new) candidate hidden state (n);
- gray - the (old) hidden state (h).

GRU Cell in Code

```
n_features = 2
hidden_dim = 2
torch.manual_seed(17)
gru_cell = nn.GRUCell(input_size=n_features,
                      hidden_size=hidden_dim)
gru_state = gru_cell.state_dict()
gru_state

# We can get the state matrices through the keys
Wi, bi = gru_state['weight_ih'], gru_state['bias_ih']
Wh, bh = gru_state['weight_hh'], gru_state['bias_hh']

# We can use split to get tensors for each component
Wir, Wiz, Win = Wi.split(hidden_dim, dim=0)
bir, biz, bin = bi.split(hidden_dim, dim=0)
Whr, Whz, Whn = Wh.split(hidden_dim, dim=0)
bhr, bhz, bhn = bh.split(hidden_dim, dim=0)
```

code/gru_cell.py

GRU Cell State

```
OrderedDict([('weight_ih',
              tensor([[-0.0930,  0.0497],
                      [ 0.4670, -0.5319],
                      [-0.6656,  0.0699],
                      [-0.1662,  0.0654],
                      [-0.0449, -0.6828],
                      [-0.6769, -0.1889]])),
             ('weight_hh',
              tensor([[-0.4167, -0.4352],
                      [-0.2060, -0.3989],
                      [-0.7070, -0.5083],
                      [ 0.1418,  0.0930],
                      [-0.5729, -0.5700],
                      [-0.1818, -0.6691]])),
             ('bias_ih',
              tensor([-0.4316,  0.4019,  0.1222, -0.4647, -0.5578,  0.4493])),
             ('bias_hh',
              tensor([-0.6800,  0.4422, -0.3559, -0.0279,  0.6553,  0.2918]))])
```

$$W_{ir} = \begin{cases} -0.0930, & 0.0497, \\ 0.4670, & -0.5319, \end{cases}$$

$$W_{iz} = \begin{cases} -0.6656, & 0.0699, \\ -0.1662, & 0.0654, \end{cases}$$

$$W_{in} = \begin{cases} -0.0449, & -0.6828, \\ -0.6769, & -0.1889 \end{cases}$$

$$\underbrace{-0.4316, 0.4019}_{b_{ir}}, \underbrace{0.1222, -0.4647}_{b_{iz}}, \underbrace{-0.5578, 0.4493}_{b_{in}}$$

Instead of returning separate weights for each one of GRU cell's components (r, z, and n), the `state_dict` returns the concatenated weights and biases

The shape is

- `(3*hidden_dim, n_features)` for `weight_ih`,
- `(3*hidden_dim, hidden_dim)` for `weight_hh`, and
- `(3*hidden_dim)` for both biases.

Manually Replicating a GRU Cell - Step 1

Create six Linear Layers by loading state dictionary, for

- input to reset, hidden to reset;
- input to update, hidden to update; and
- input to (new) candidate hidden, (old) hidden to (new) hidden

```
def linear_layers(Wi, bi, Wh, bh):
    hidden_dim, n_features = Wi.size()
    lin_input = nn.Linear(n_features, hidden_dim)
    lin_input.load_state_dict({'weight': Wi, 'bias': bi})
    lin_hidden = nn.Linear(hidden_dim, hidden_dim)
    lin_hidden.load_state_dict({'weight': Wh, 'bias': bh})
    return lin_hidden, lin_input

# reset gate - red
r_hidden, r_input = linear_layers(Wir, bir, Whr, bhr)
# update gate - blue
z_hidden, z_input = linear_layers(Wiz, biz, Whz, bhz)
# candidate state - black
n_hidden, n_input = linear_layers(Win, bin, Whn, bhn)
```

code/manual_gru_cell_s1.py

Manually Replicating a GRU Cell - Step 2

Pass the input and hidden state through these layers:

```
def reset_gate(h, x):
    thr = r_hidden(h)
    txr = r_input(x)
    r = torch.sigmoid(thr + txr)
    return r  # red
def update_gate(h, x):
    thz = z_hidden(h)
    txz = z_input(x)
    z = torch.sigmoid(thz + txz)
    return z  # blue
def candidate_n(h, x, r):
    thn = n_hidden(h)
    txn = n_input(x)
    n = torch.tanh(r * thn + txn)
    return n  # black
# first_corner is one data point of the input sequence
# initial hidden state is set to be all zeros
r = reset_gate(initial_hidden, first_corner)
z = update_gate(initial_hidden, first_corner)
n = candidate_n(initial_hidden, first_corner, r)
h_prime = n*(1-z) + initial_hidden*z
```

Results are the Same as `nn.GRUCell()`

```
n_features = 2
hidden_dim = 2
torch.manual_seed(17)
gru_cell = nn.GRUCell(input_size=n_features,
                      hidden_size=hidden_dim)
gru_cell(first_corner)
```

code/gru_cell_applied.py

GRU Layer

The `nn.GRU` layer takes care of the hidden state handling for us, no matter how long the input sequence is.

GRU Layer is similar to the RNN layer in PyTorch code

- The arguments, inputs, and outputs are almost exactly the same for both of them, **except for one small difference: you cannot choose a different activation function anymore**.
- You can create stacked GRUs and bidirectional GRUs as well. The logic doesn't change a bit - the only difference is that you'll be using a fancier GRU cell instead of the basic RNN cell.

Long Short Term Memory (LSTM)

LSTM - Two State instead of One

Hidden State and Cell State

- The regular **hidden state (h)**, which is bounded by the hyperbolic tangent, as usual, corresponding to the **short-term memory**.
- A **cell state (c)**, which is unbounded, corresponding to the **long-term memory**.

1. Use a regular RNN to generate a candidate hidden state (g):

$$g = tanh(t_{hg} + t_{xg})$$

2. The new cell state (c') is computed using a weighted sum of the old cell state and the candidate hidden state (g), where i is the input gate and f is the forget gate:

$$c' = g * i + c * f$$

3. The new hidden state (h') is obtained by first bounding the cell state before applying an output gate.

$$h' = tanh(c') * o$$

THE UNIVERSITY OF WESTERN AUSTRALIA

LSTM Equations

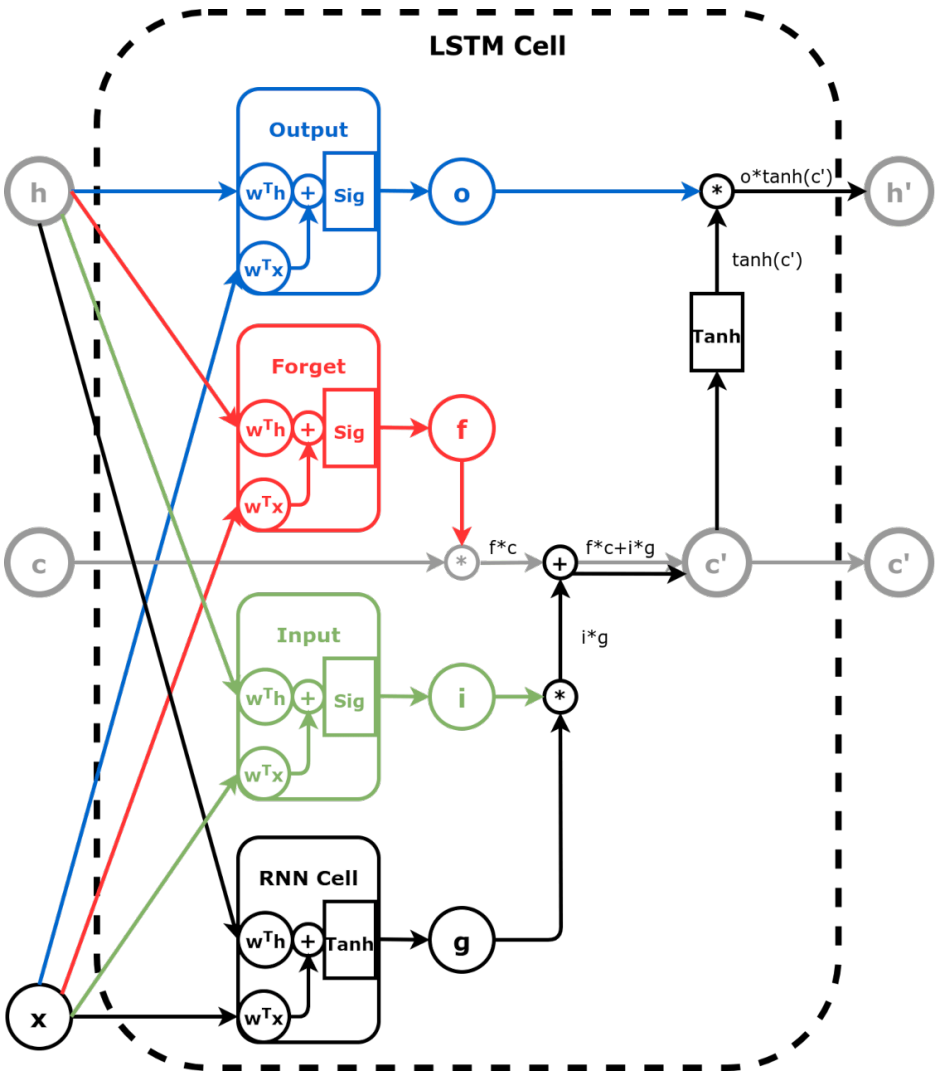
RNN vs. GRU vs. LSTM

$$
\begin{aligned}
RNN: h' &= tanh(t_h + t_x) \\
GRU: h' &= tanh(r * t_{hn} + t_{xn}) * (1 - z) + h * z \\
LSTM: c' &= \underbrace{tanh(t_{hg} + t_{xg})}_{g} * i + c * f \\
h' &= tanh(c') * o
\end{aligned}
$$

LSTM Gates

$$
\begin{aligned}
i(nput\ gate) &= \sigma(t_{hi} + t_{xi}) \\
f(orget\ gate) &= \sigma(t_{hf} + t_{xf}) \\
o(utput\ gate) &= \sigma(t_{ho} + t_{xo}) \\
g &= tanh(t_{hg} + t_{xg})
\end{aligned}
$$

Gates' internal transformation



$$
\begin{aligned}
i\ (hidden):\ t_{hi} &= W_{hi}\ h + b_{hi} \\
i\ (input):\ t_{xi} &= W_{ii}\ x + b_{ii} \\
f\ (hidden):\ t_{hf} &= W_{hf}\ h + b_{hf} \\
f\ (input):\ t_{xf} &= W_{if}\ x + b_{if} \\
g\ (hidden):\ t_{hg} &= W_{hg}\ h + b_{hg} \\
g\ (input):\ t_{xg} &= W_{ig}\ x + b_{ig} \\
o\ (hidden):\ t_{ho} &= W_{ho}\ h + b_{ho} \\
o\ (input):\ t_{xo} &= W_{io}\ x + b_{io}
\end{aligned}
$$

Understanding LSTM

1. Learn to literally ignore the internals of the gates: o, f, and i are simply values between zero and one (for each dimension).
2. Pretend $i = 1$ and $f = 0$, the new cell state c' is then equivalent to the output of a simple RNN.
3. Pretend $i = 0$ and $f = 1$, the new cell state c' is simply a copy of the old cell state (in other words, the data (x) does not have any effect).
4. If we decrease o all the way to zero, the new hidden state h' is going to be zero as well.

Recall that the gradient of `tanh` gets very small really fast!

NO `tanh` for Cell State to combat Vanishing Gradients

The cell state is computed using two multiplications and one addition only. **Cell state does not suffer from gradient vanishing - a "highway for gradients"**!

LSTM Cell in PyTorch

```
n_features = 2
hidden_dim = 2

torch.manual_seed(17)
lstm_cell = nn.LSTMCell(input_size=n_features, hidden_size=
    hidden_dim)
lstm_state = lstm_cell.state_dict()

Wx, bx = lstm_state['weight_ih'], lstm_state['bias_ih']
Wh, bh = lstm_state['weight_hh'], lstm_state['bias_hh']

# Split weights and biases for data points
Wxi, Wxf, Wxg, Wxo = Wx.split(hidden_dim, dim=0)
bxi, bxf, bxg, bxo = bx.split(hidden_dim, dim=0)
# Split weights and biases for hidden state
Whi, Whf, Whg, Who = Wh.split(hidden_dim, dim=0)
bhi, bhf, bhg, bho = bh.split(hidden_dim, dim=0)
# Creates linear layers for the components
i_hidden, i_input = linear_layers(Wxi, bxi, Whi, bhi)
f_hidden, f_input = linear_layers(Wxf, bxf, Whf, bhf)
o_hidden, o_input = linear_layers(Wxo, bxo, Who, bho)
```

code/lstm_cell.py

LSTM Cell State

```
OrderedDict([('weight_ih',
              tensor([[-0.0930,  0.0497],
                      [ 0.4670, -0.5319],
                      [-0.6656,  0.0699],
                      [-0.1662,  0.0654],
                      [-0.0449, -0.6828],
                      [-0.6769, -0.1889],
                      [-0.4167, -0.4352],
                      [-0.2060, -0.3989]])),
             ('weight_hh',
              tensor([[-0.7070, -0.5083],
                      [ 0.1418,  0.0930],
                      [-0.5729, -0.5700],
                      [-0.1818, -0.6691],
                      [-0.4316,  0.4019],
                      [ 0.1222, -0.4647],
                      [-0.5578,  0.4493],
                      [-0.6800,  0.4422]])),
             ('bias_ih',
              tensor([-0.3559, -0.0279,  0.6553,  0.2918,
0.4007,  0.3262, -0.0778, -0.3002])),
             ('bias_hh',
              tensor([-0.3991, -0.3200,  0.3483, -0.2604,
-0.1582,  0.5558,  0.5761, -0.3919]))])
```

Sequence to Sequence

Seq2Seq

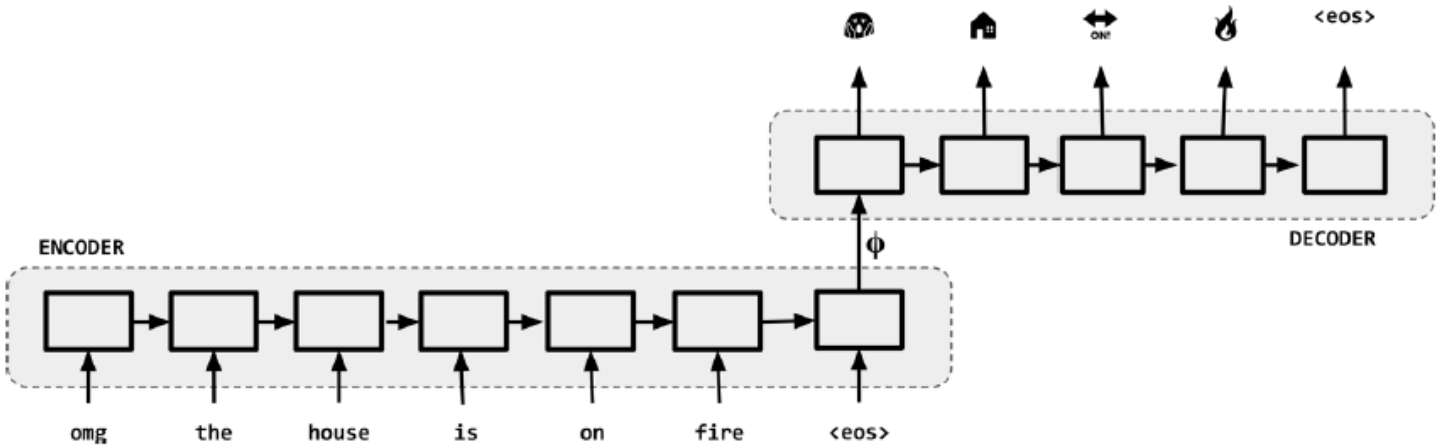
Sequence-to-sequence (Seq2Seq) problems are more complex than those we handled so far. There are two sequences now:

- the **source** sequence, and
- the **target** sequence

We use the source sequence to predict the target sequence, and they may even have different lengths.

- Seq2Seq models and their variations are dominating the fields of **Machine Translation**.
- Seq2Seq models are a special case of a general family of models called **encoder-decoder models**.

Seq2Seq Illustration



English to Emojis translation using a Seq2Seq model

Encoder-Decoder Architecture

Encoder-Decoder Model

An Encoder-Decoder Model is a composition of two models, an "encoder" and a "decoder", that are typically jointly trained. The encoder model takes an input and produces an encoding or a representation (ψ) of the input, which is usually a vector.

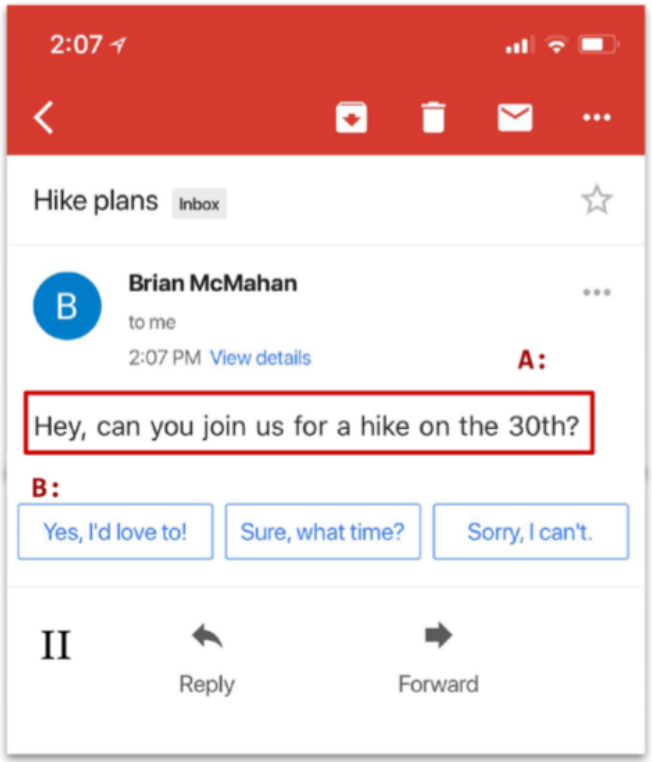
- The encoder's goal is to generate a representation of the source sequence, that is, to encode it.
- The decoder's goal is to generate the target sequence from an initial representation, that is, to decode it.

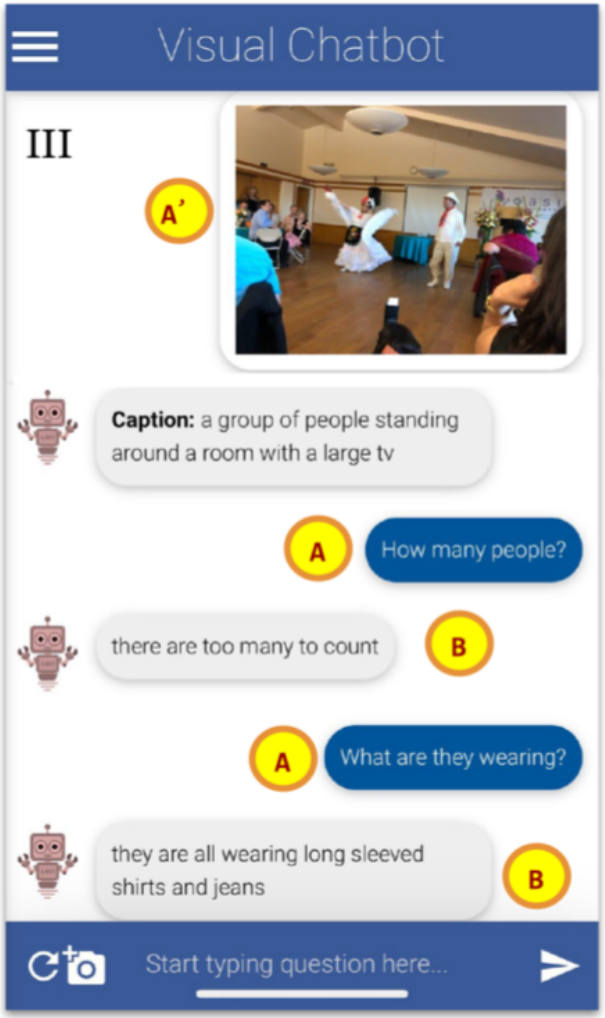
Seq2Seq models are thus defined as encoder-decoder models in which the encoder and decoder are sequence models and the inputs and outputs are both sequences, possibly of different lengths.

Can you think of other encoder-decoder modes that are **not** necessarily seq2seq?

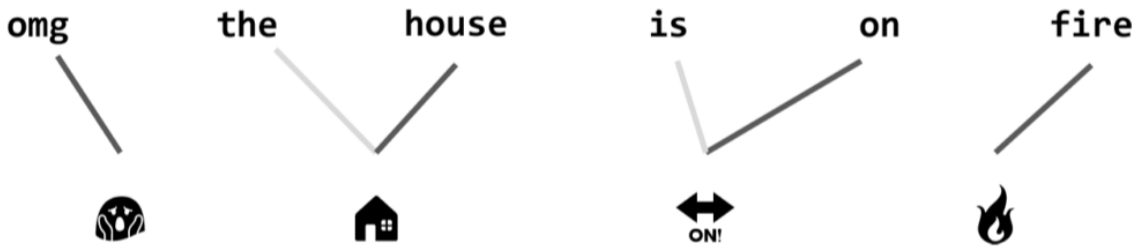
THE UNIVERSITY OF WESTERN AUSTRALIA

Seq2Seq Real Word Applications





Machine Translation - Very Briefly



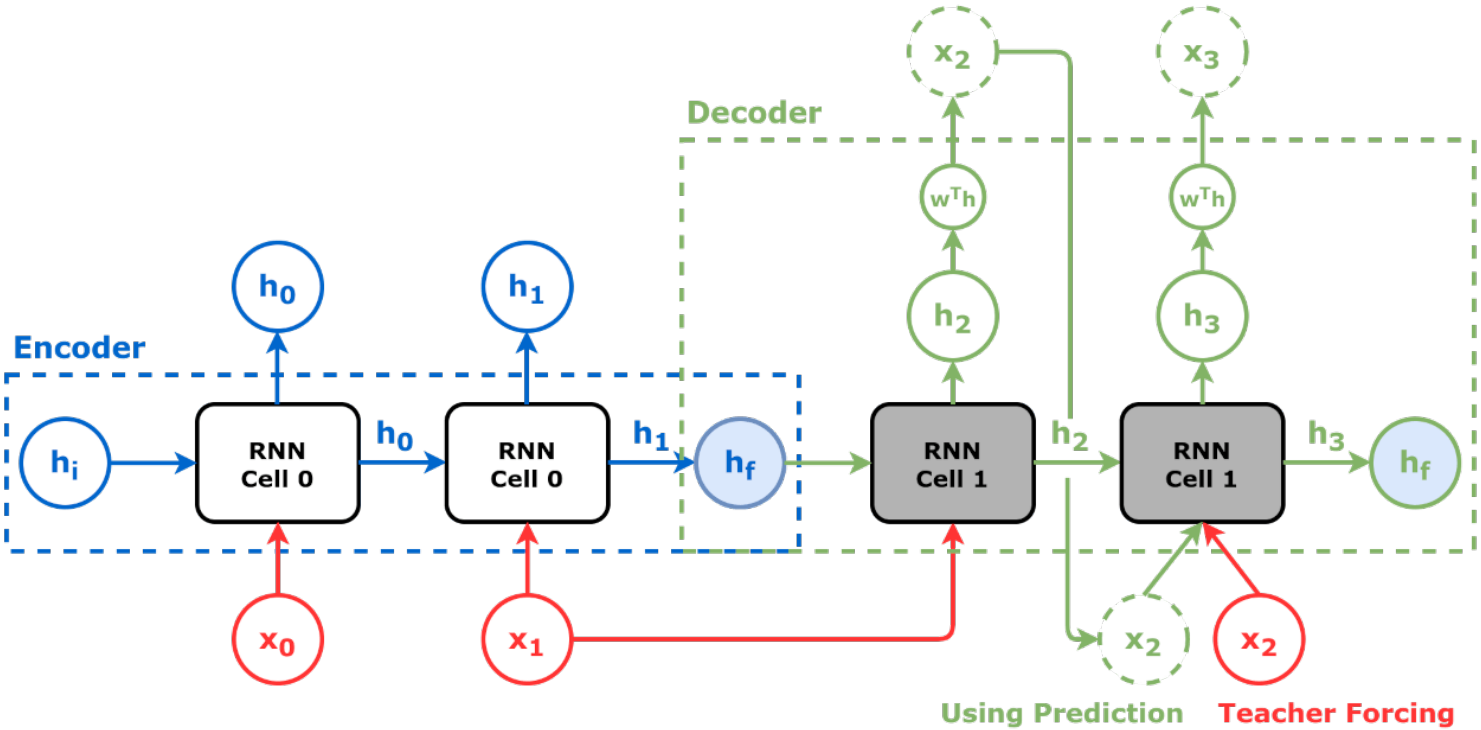
An intelligent smart phone keyboard translates typed English sentences to Emojis.

Traditionally, solutions to S2S problems were attempted by using linguistic and heuristic heavy statistical approaches. For example, the above will at least involve

- tokenisation of source sequence,
- a dictionary that translates each token into one or more target token, and
- align the target tokens in the target language to meet both syntactic and semantic constraints.

[See Statistical Machine Translation] for traditional *phrased based statistical approaches* in machine translation.

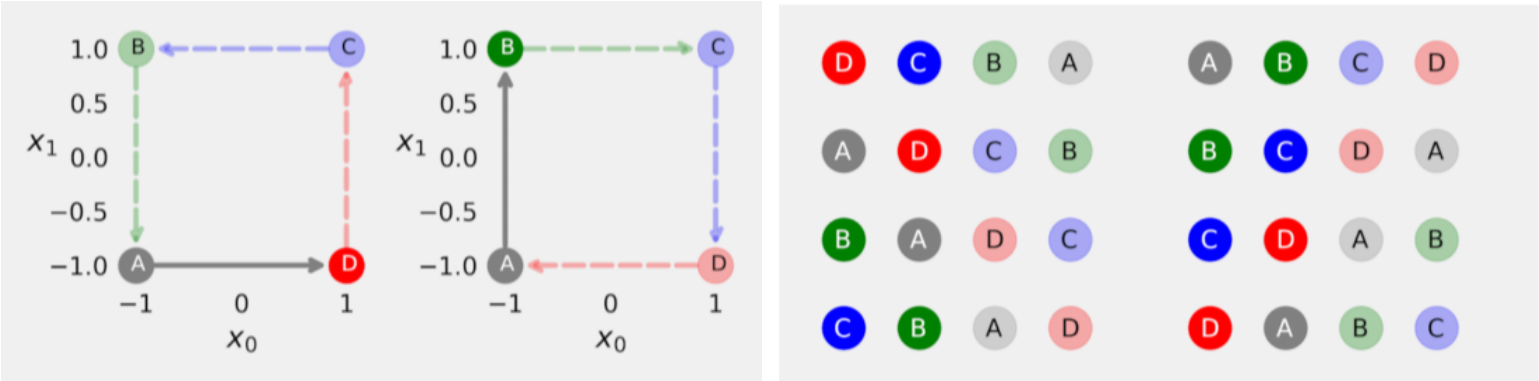
How does an Encoder-Decoder Model for Seq2Seq look like?



One RNN as the Encoder, another RNN as the Decoder

You can use any RNN as an encoder and an decoder, be it an Elman RNN, LSTM, or GRU.

Sequence Prediction Example using Seq2Seq



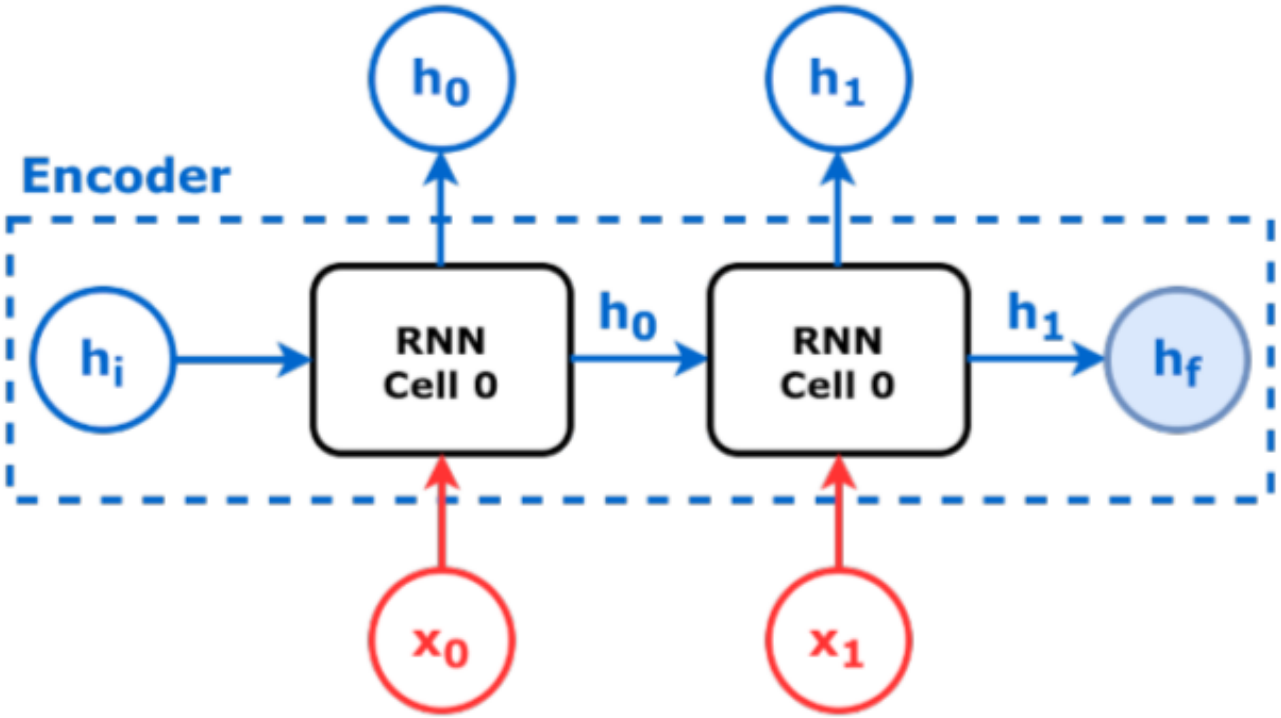
Left: Synthetic squares with Directions; **Right**: All eight possible sequences

Task:

Given two starting points (e.g. A, B) predicting the next two points.

The encoder through training "should" encode direction information of the two data points, thus predicting the next two data points in that direction.

Encoder



A simple encoder in PyTorch

```python
class Encoder(nn.Module):
    def __init__(self, n_features, hidden_dim):
        super().__init__()
        self.hidden_dim = hidden_dim
        self.n_features = n_features
        self.hidden = None
        self.basic_rnn = nn.GRU(self.n_features, self.hidden_dim
    , batch_first=True)

    def forward(self, X):
        rnn_out, self.hidden = self.basic_rnn(X)

        return rnn_out # N, L, F
```

code/encoder.py

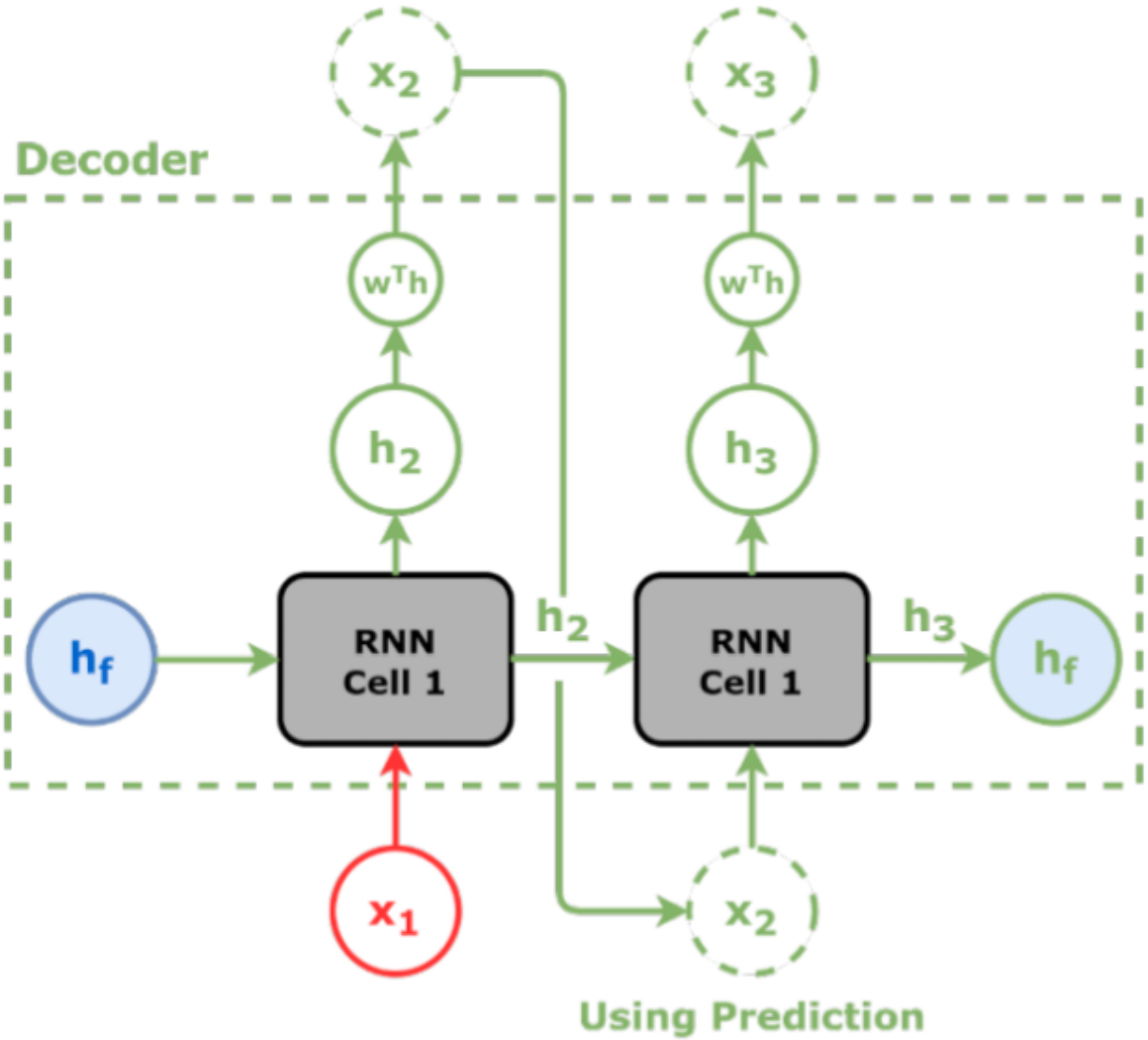
Encode a sequence

```
full_seq = torch.tensor([[-1, -1], [-1, 1], [1, 1], [1, -1]]).
    float().view(1, 4, 2)
source_seq = full_seq[:, :2] # first two corners
target_seq = full_seq[:, 2:] # last two corners

torch.manual_seed(21)
encoder = Encoder(n_features=2, hidden_dim=2)
# output is N, L, F
hidden_seq = encoder(source_seq)
# takes last hidden state
hidden_final = hidden_seq[:, -1:]
hidden_final
```

code/encode_data_point.py

Decoder


Decoder
x2
x3
wTh
wTh
h2
h3
hf
RNN Cell 1
h2
RNN Cell 1
h3
hf
x1
x2
Using Prediction

How does the Encoder work?

1. The initial hidden state the encoder's final hidden state (h_f in blue), and the first input is the last data point in the source sequence (x_1 in red).
2. The first cell will output a new hidden state (h_2): that's both the output of that cell and one of the inputs of the next cell, as we've already seen in RNN, GRU and LSTM.
3. Before, we'd only run the final hidden state through a linear layer to produce the logits for **classification**, but now we'll run the output of every cell through a linear layer ($w^T h$) to convert each hidden state into predicted coordinates (x_2 and x_3)
4. The predicted coordinates are then used as one of the inputs of the second step (x_2).

Decoder Coded in PyTorch I

```
class Decoder(nn.Module):
    def __init__(self, n_features, hidden_dim):
        super().__init__()
        self.hidden_dim = hidden_dim
        self.n_features = n_features
        self.hidden = None
        self.basic_rnn = nn.GRU(self.n_features,
                self.hidden_dim, batch_first=True)
        self.regression = nn.Linear(self.hidden_dim,
                self.n_features)

    def init_hidden(self, hidden_seq):
        # We only need the final hidden state
        # N, 1, H
        hidden_final = hidden_seq[:, -1:]
        # But we need to make it sequence-first
        # 1, N, H
        self.hidden = hidden_final.permute(1, 0, 2)

    def forward(self, X):
        # X is N, 1, F
```

Decoder Coded in PyTorch II

```
22          batch_first_output, self.hidden =
23                          self.basic_rnn(X, self.hidden)
24
25          last_output = batch_first_output[:, -1:]
26          out = self.regression(last_output)
27
28          # N, 1, F
29          return out.view(-1, 1, self.n_features)
```

code/decoder.py

Hidden State as an Attribute of Decoder Model

A small, yet fundamental, difference between encoder and decoder

Since the decoder uses the prediction of one step as input to the next, we'll have to manually loop over the generation of the target sequence.

- This also means we need to keep track of the hidden state from one step to the next, using the hidden state of one step as input to the next as well.
- But, instead of making the hidden state both an input and an output of the forward method, we can easily (and quite elegantly) handle this by making the hidden state an attribute of our decoder model.

The Decoder's First Data Point

In the simplified Square Sequence predication example,

- the decoder's first data point is actually the last data point in the source sequence;
- this is because the target sequence is not a new sequence, but the continuation of the source sequence.

This is not always the case: in some Natural Language Processing tasks, like translation,

- the target sequence is a new sequence;
- the first data point is therefore some "special" token that indicates the start of that new sequence.

Decode with Teacher-Forcing

```
# Initial hidden state is encoder's final hidden state
decoder.init_hidden(hidden_seq)
# Initial data point is the last element of
# source sequence
inputs = source_seq[:, -1:]

teacher_forcing_prob = 0.5
target_len = 2
for i in range(target_len):
    print(f'Hidden: {decoder.hidden}')
    out = decoder(inputs)
    print(f'Output: {out}\n')
    # If it is teacher forcing
    if torch.rand(1) <= teacher_forcing_prob:
        # Takes the actual element
        inputs = target_seq[:, i:i+1]
    else:
        # Otherwise uses the last predicted output
        inputs = out
```

code/decode_to_a_sequence.py

Put everything together - EncoderDecoder I

```
class EncoderDecoder(nn.Module):
    def __init__(self, encoder, decoder, input_len, target_len,
    teacher_forcing_prob=0.5):
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.input_len = input_len
        self.target_len = target_len
        self.teacher_forcing_prob = teacher_forcing_prob
        self.outputs = None

    def init_outputs(self, batch_size):
        device = next(self.parameters()).device
        # N, L (target), F
        self.outputs = torch.zeros(batch_size,
            self.target_len,
            self.encoder.n_features).to(device)

    def store_output(self, i, out):
        # Stores the output
        self.outputs[:, i:i+1, :] = out
```

Put everything together - EncoderDecoder II

```
21
22      def forward(self, X):
23          # splits the data in source and target sequences
24          # the target seq will be empty in testing mode
25          # N, L, F
26          source_seq = X[:, :self.input_len, :]
27          target_seq = X[:, self.input_len:, :]
28          self.init_outputs(X.shape[0])
29
30          # Encoder expected N, L, F
31          hidden_seq = self.encoder(source_seq)
32          # Output is N, L, H
33          self.decoder.init_hidden(hidden_seq)
34
35          # The last input of the encoder is also
36          # the first input of the decoder
37          dec_inputs = source_seq[:, -1:, :]
38
39          # Generates as many outputs as the target length
40          for i in range(self.target_len):
41              # Output of decoder is N, 1, F
```

Put everything together - EncoderDecoder III

```
42              out = self.decoder(dec_inputs)
43              self.store_output(i, out)
44
45              prob = self.teacher_forcing_prob
46              # In evaluation/test the target sequence is
47              # unknown, so we cannot use teacher forcing
48              if not self.training:
49                  prob = 0
50
51              # If it is teacher forcing
52              if torch.rand(1) <= prob:
53                  # Takes the actual element
54                  dec_inputs = target_seq[:, i:i+1, :]
55              else:
56                  # Otherwise uses the last predicted output
57                  dec_inputs = out
58
59          return self.outputs
```

code/encoder_decoder.py

Take Aways

- the basic gating mechanisms of GRU and LSTM, and what problems they try to address
- the foundation of a sequence to sequence model in terms of an encoder and decoder architecture

References

[1] Daniel Voigt Godoy. **Deep Learning with PyTorch Step-by-Step: A Beginner's Guide**. Published at: http://leanpub.com/pytorch, Github Repo: https://github.com/dvgodoy/PyTorchStepByStep, 2021.

[2] Philipp Koehn. **Statistical machine translation**. Cambridge University Press, Cambridge, 2012 - 2010. ISBN 9780521874151.

[3] Delip Rao. **Natural language processing with PyTorch : build intelligent language applications using deep learning**. O'Reilly Media, Beijing, first edition. edition, 2019. ISBN 9781491978207.