Lab 6: More on RNNs

University of Washington

ECE 596/AMATH 563

Spring 2021

Outline

Part 1: Gated RNN Architectures

- LSTM
- GRU

Part 2: Examples of Different RNN Problems

- One-to-one
- One-to-many
- Many-to-one
- Many-to-many

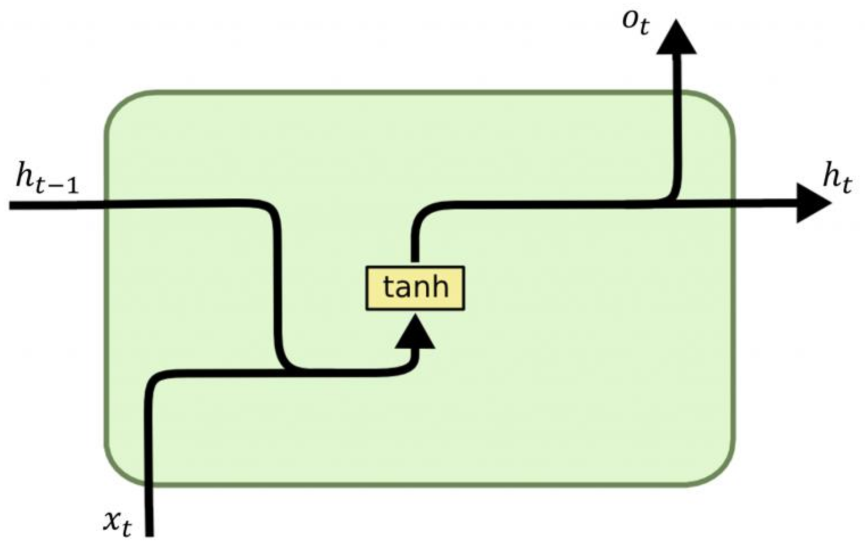
Part 3: Other RNN Variants

- Deep RNN
- Bidirectional RNN

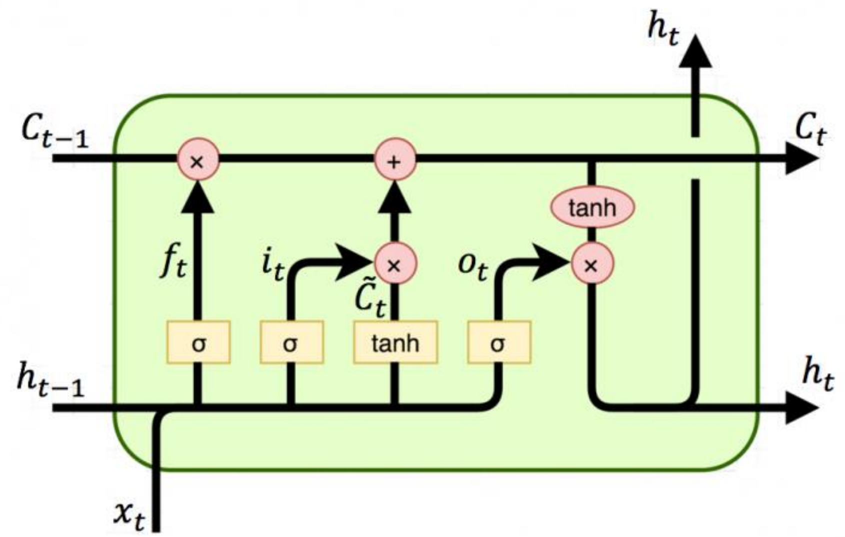
Lab Assignment

Part 1: Gated RNN Architectures

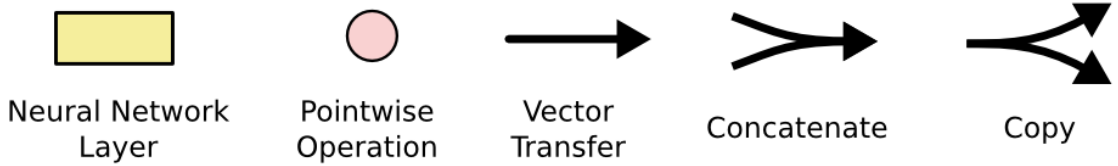
LSTM (Long Short-Term Memory)



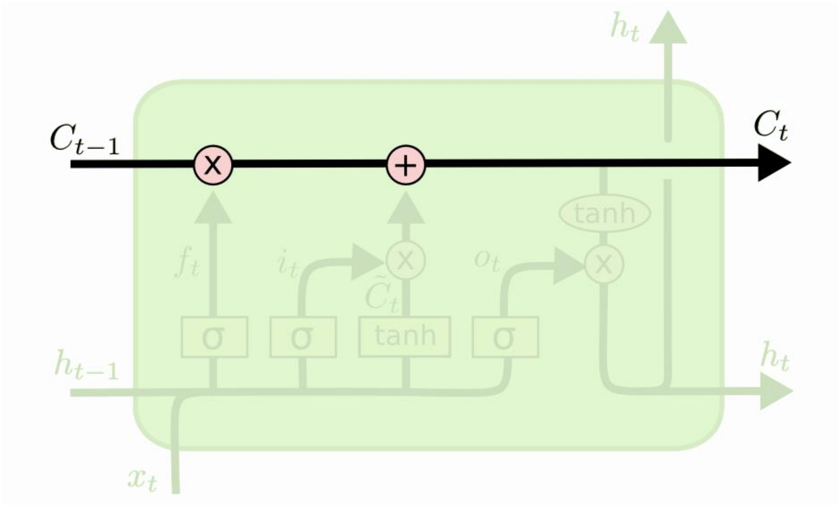
Vanilla RNN



LSTM



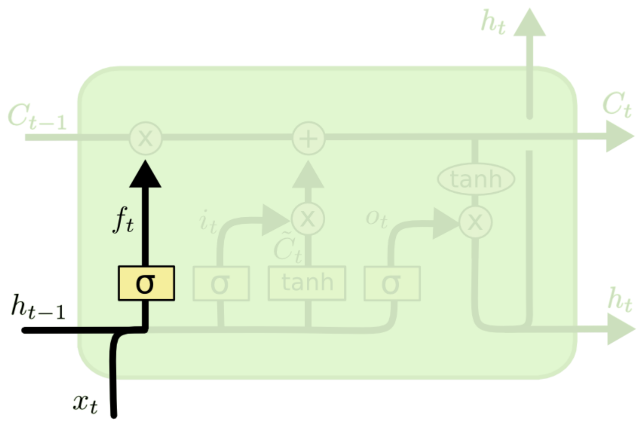
LSTM: Detailed Architecture



Cell state

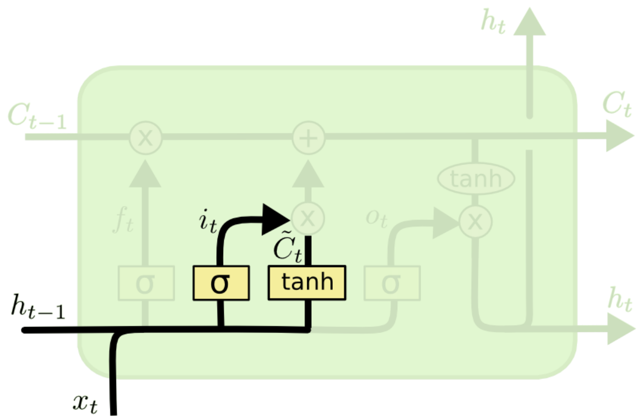
- Unique to LSTM
- Long term memory of the model

LSTM: Detailed Architecture



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$

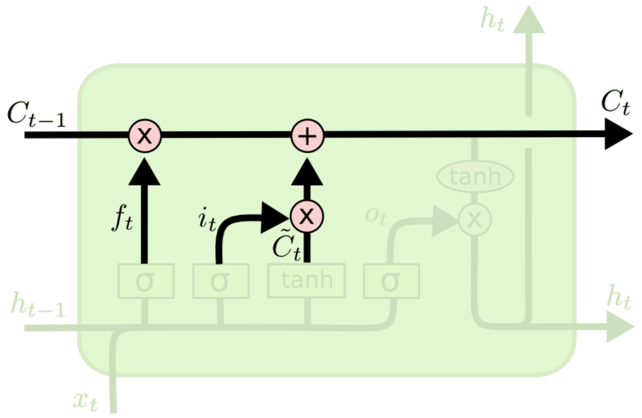
Forget gate layer



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

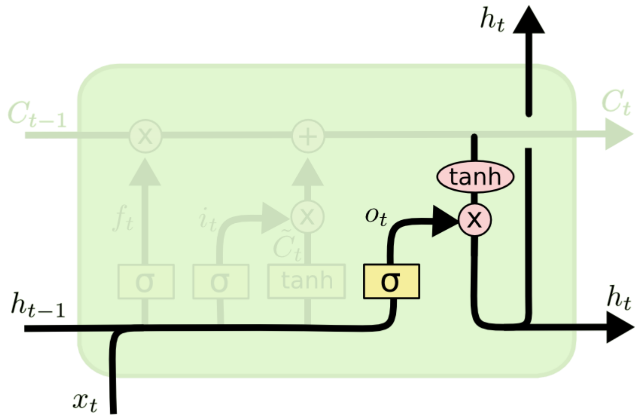
Input gate layer

LSTM: Detailed Architecture



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Update cell state



$$o_t = \sigma\left(W_o \left[h_{t-1}, x_t\right] + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Output gate layer

LSTM: PyTorch Implementation

```
class LSTMModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, layer_dim, output_dim):
        super(LSTMModel, self).__init__()

        self.hidden_dim = hidden_dim

        self.layer_dim = layer_dim

        self.lstm = nn.LSTM(input_dim, hidden_dim, layer_dim)

        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        h0 = torch.zeros(self.layer_dim, x.size(0), self.hidden_dim).requires_grad_()

        c0 = torch.zeros(self.layer_dim, x.size(0), self.hidden_dim).requires_grad_()

        out, (hn, cn) = self.lstm(x, (h0, c0))

        out = self.fc(out[:, -1, :])
        return out
```

Hidden layer dimension

# of hidden layers

 # PyTorch built in LSTM architecture

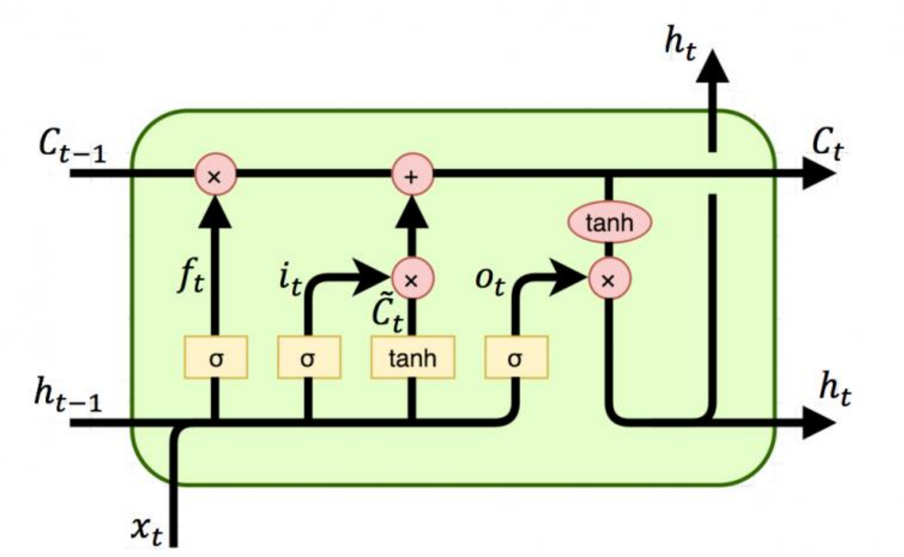
Readout layer

Initialize hidden/cell states

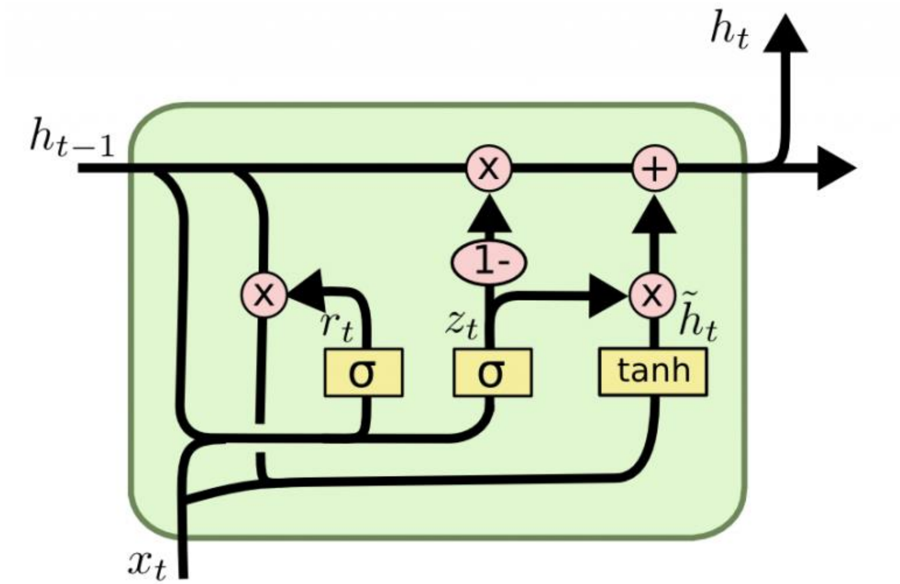
Pass the input x to LSTM
(x.shape = seq_len, batch, input_size)

Pass the hidden state of last time step to readout layer

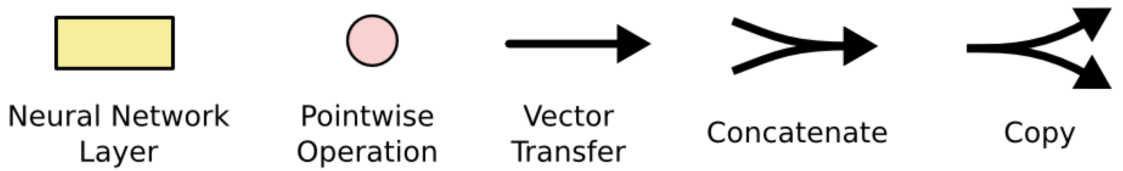
GRU (Gated Recurrent Units)



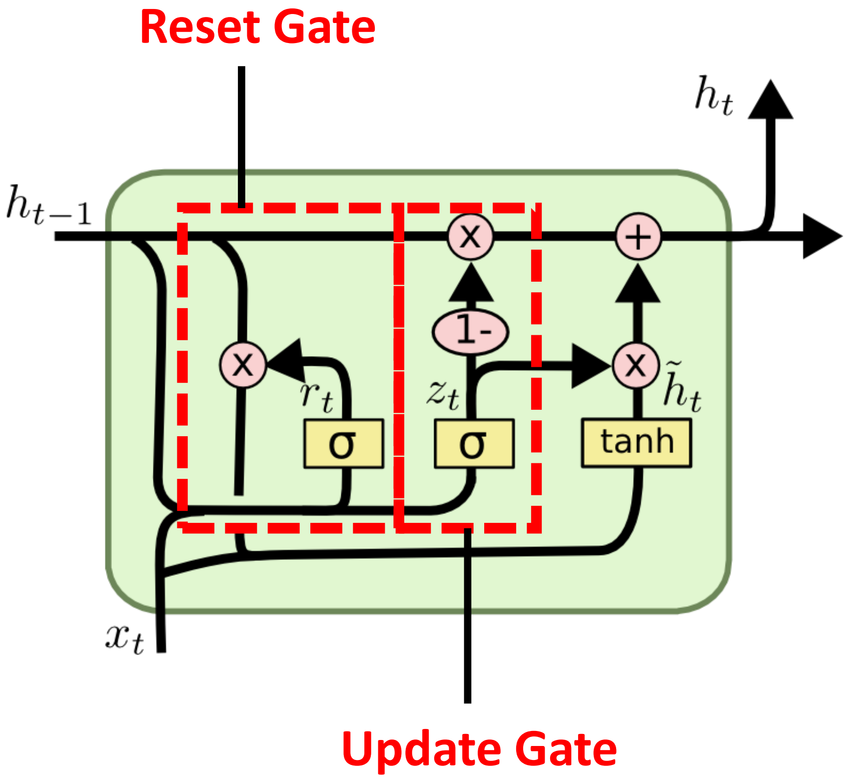
LSTM



GRU



GRU: Detailed Architecture



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

GRU: PyTorch Implementation

```
class GRUModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, num_layers, output_dim):
        super(LSTMModel, self).__init__()

        self.hidden_dim = hidden_dim

        self.num_layers = num_layers

        self.gru = nn.GRU(input_dim, hidden_dim, num_layers)

        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        h0 = torch.zeros(self.layer_dim, x.size(0), self.hidden_dim).requires_grad_()

        out, (hn, cn) = self.gru(x, h0)

        out = self.fc(out[:, -1, :])

        return out
```

Hidden layer dimension

# of hidden layers

 # PyTorch built in GRU architecture

Readout layer

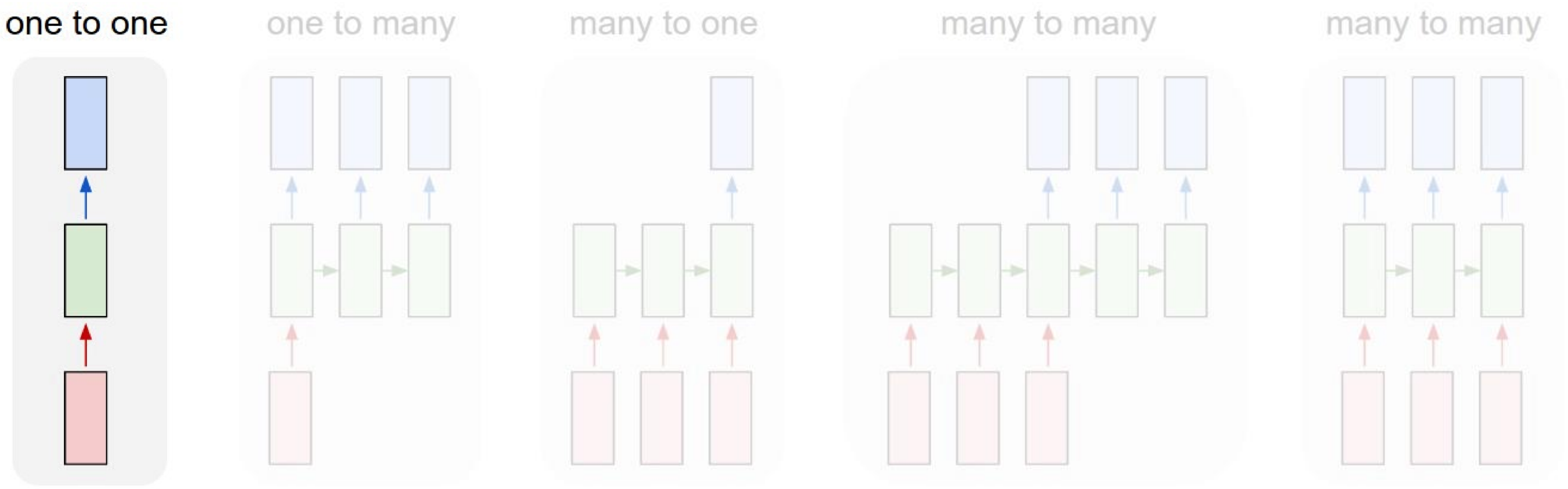
Initialize hidden/cell states

Pass the input x to GRU

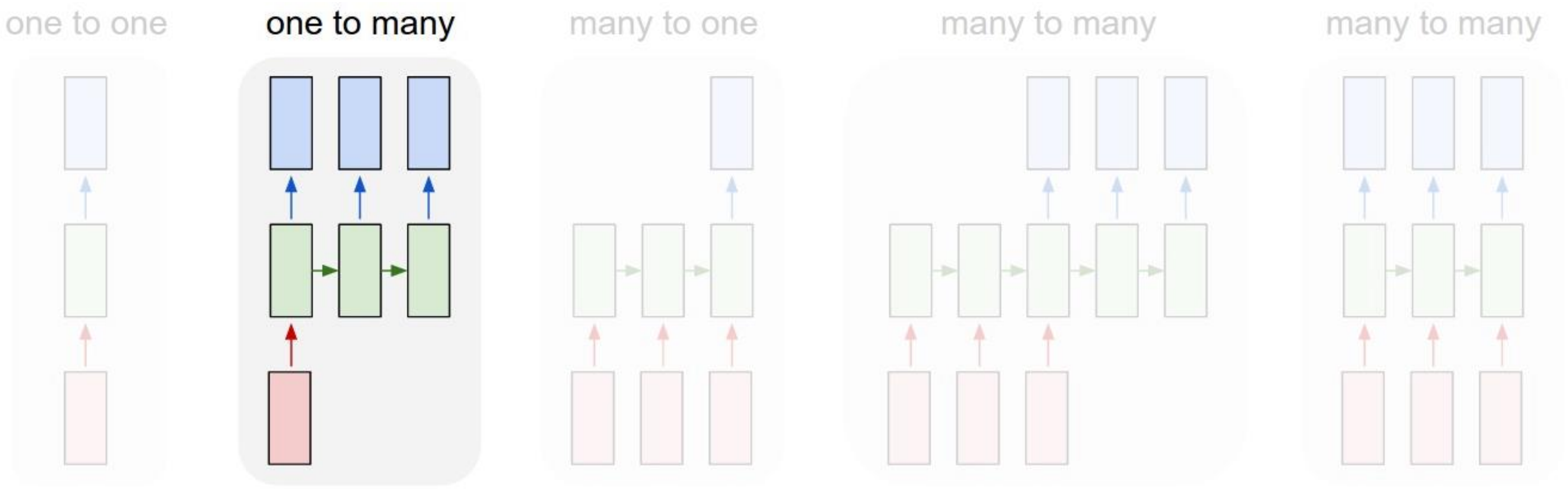
Pass the hidden state of last time step to readout layer

Part 2: Examples of Different RNN Problems

One-to-one



One-to-Many



One-to-Many

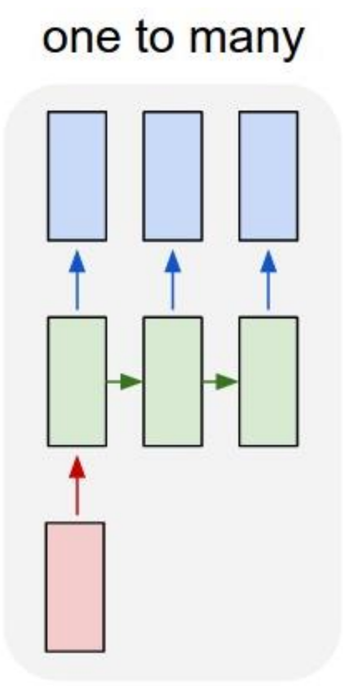
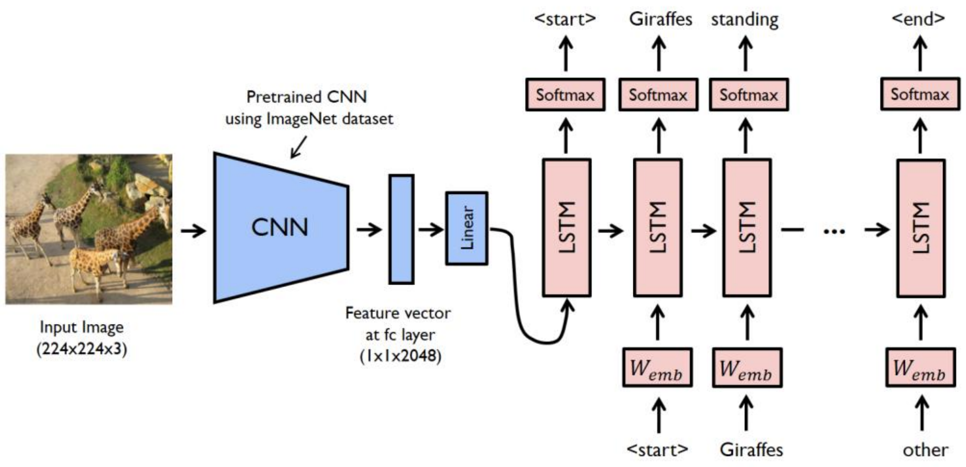
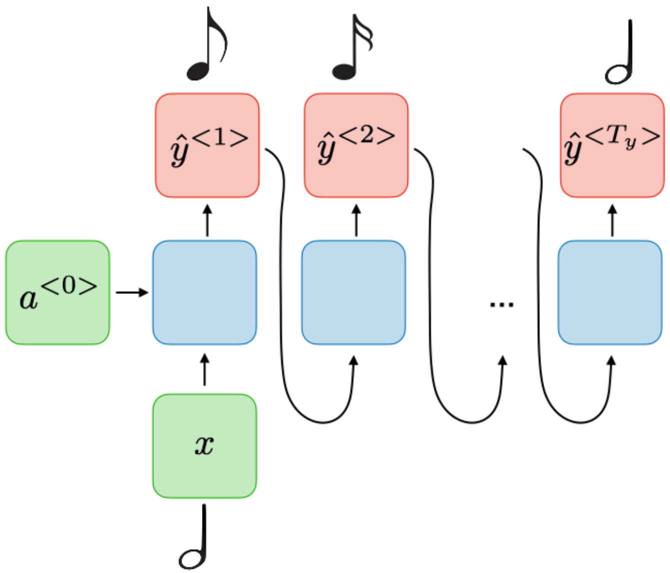


Image credit: www.analyticsvidhya.com

Image captioning

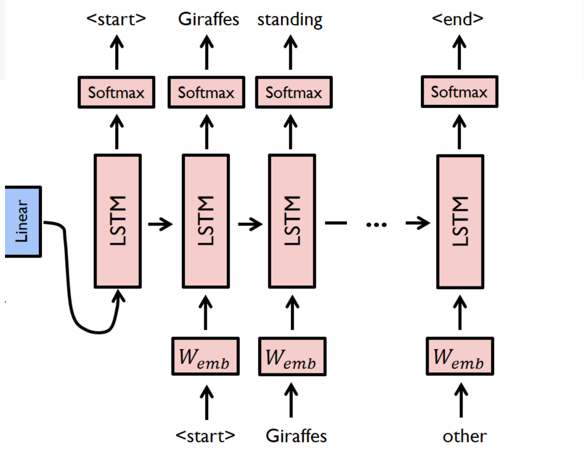


Music generation



One-to-Many: Image captioning example

```python
class DecoderRNN(nn.Module):
    def __init__(self, embed_size, hidden_size, vocab_size, num_layers=1):
        super(DecoderRNN, self).__init__()

        # define the properties
        self.embed_size = embed_size
        self.hidden_size = hidden_size
        self.vocab_size = vocab_size

        # lstm cell
        self.lstm_cell = nn.LSTMCell(input_size=embed_size, hidden_size=hidden_size)

        # output fully connected layer
        self.fc_out = nn.Linear(in_features=self.hidden_size, out_features=self.vocab_size)

        # embedding layer
        self.embed = nn.Embedding(num_embeddings=self.vocab_size, embedding_dim=self.embed_size)

        # activations
        self.softmax = nn.Softmax(dim=1)
```



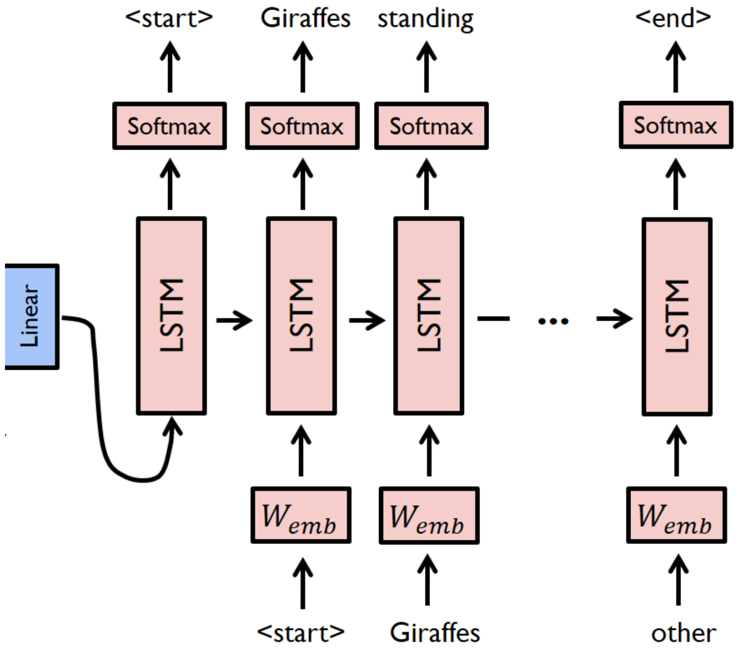
Word Embedding

Word	Integer	Vector representation
apple	1	[0 0 0 0 1]
orange	2	[0 0 0 1 0]
guava	3	[0 0 0 1 1]
blue	4	[0 0 1 0 0]
green	5	[0 0 1 0 1]
red	6	[0 0 1 1 0]

Image credit: Medium

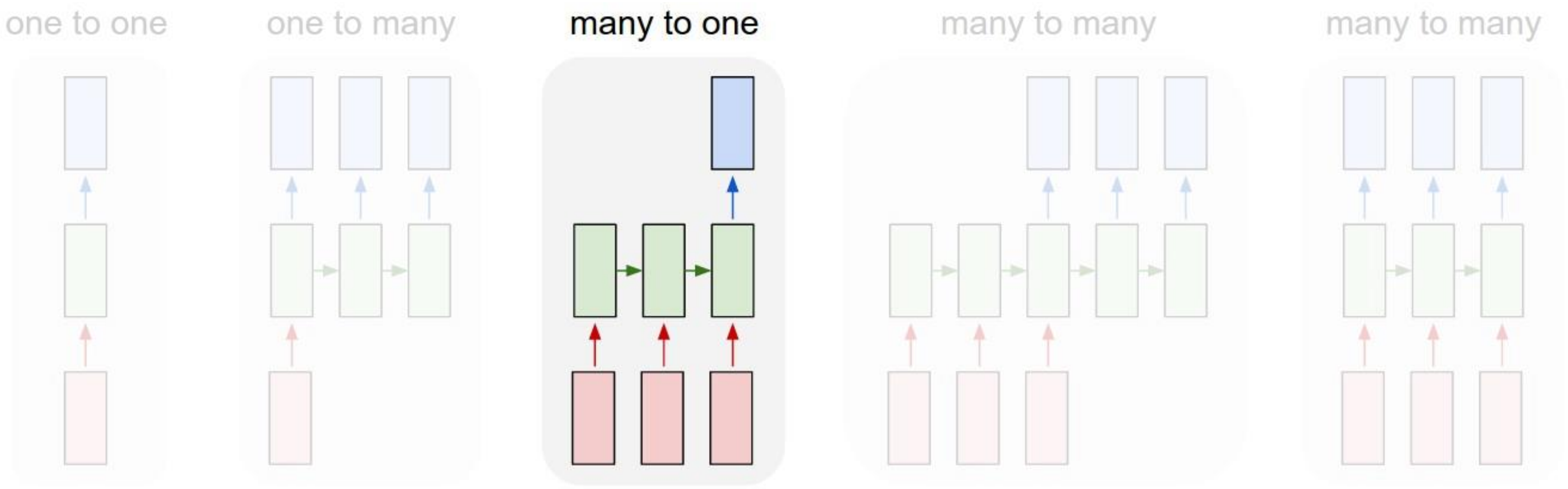
One-to-Many: Image captioning example

```python
def forward(self, features, captions):

    # batch size
    batch_size = features.size(0)

    # init the hidden and cell states to zeros
    hidden_state = torch.zeros((batch_size, self.hidden_size)).cuda()
    cell_state = torch.zeros((batch_size, self.hidden_size)).cuda()

    # define the output tensor placeholder
    outputs = torch.empty((batch_size, captions.size(1), self.vocab_size)).cuda()

    # embed the captions
    captions_embed = self.embed(captions)

    # pass the caption word by word
    for t in range(captions.size(1)):

        # for the first time step the input is the feature vector
        if t == 0:
            hidden_state, cell_state = self.lstm_cell(features, (hidden_state, cell_state))

        # for the 2nd+ time step, using teacher forcer
        else:
            hidden_state, cell_state = self.lstm_cell(captions_embed[:, t, :], (hidden_state, cell_state))

        # output of the attention mechanism
        out = self.fc_out(hidden_state)

        # build the output tensor
        outputs[:, t, :] = out

    return outputs
```

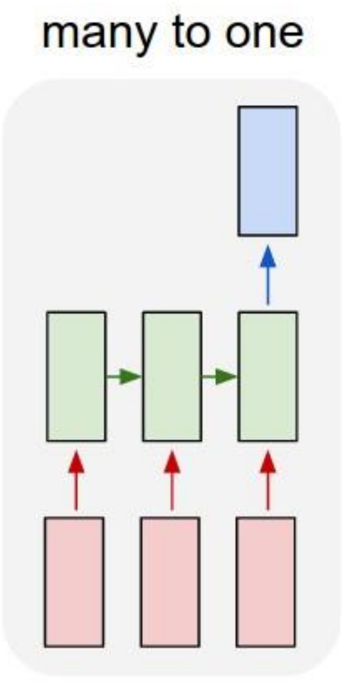


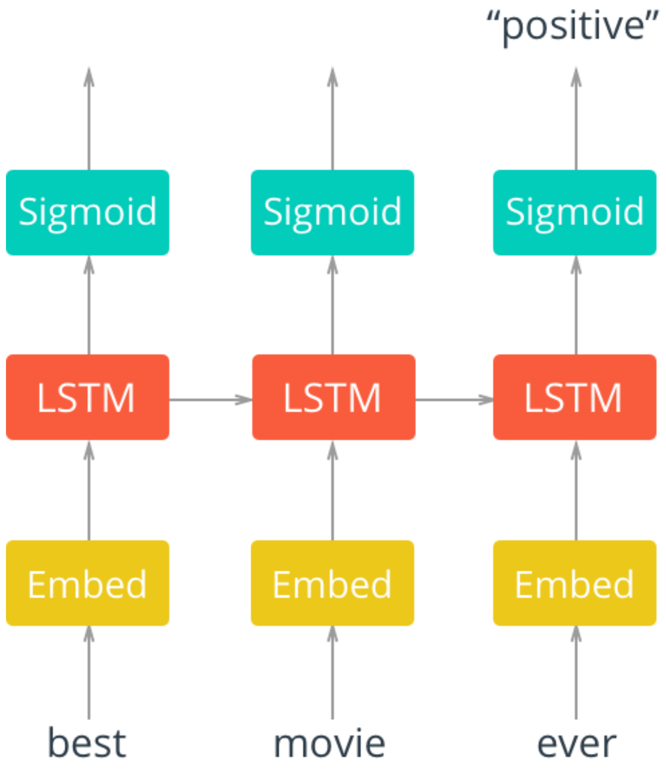
Implementation detail:
https://medium.com/@stepanulyanin/captioning-images-with-pytorch-bc592e5fd1a3

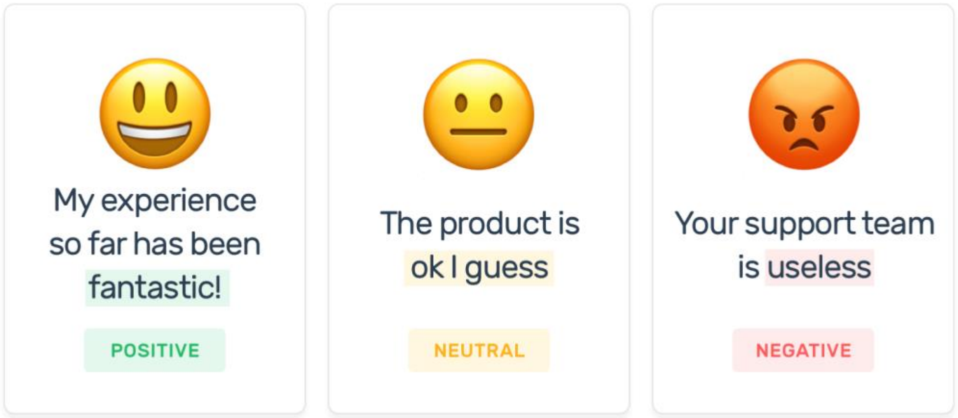
Many-to-One



Many-to-One



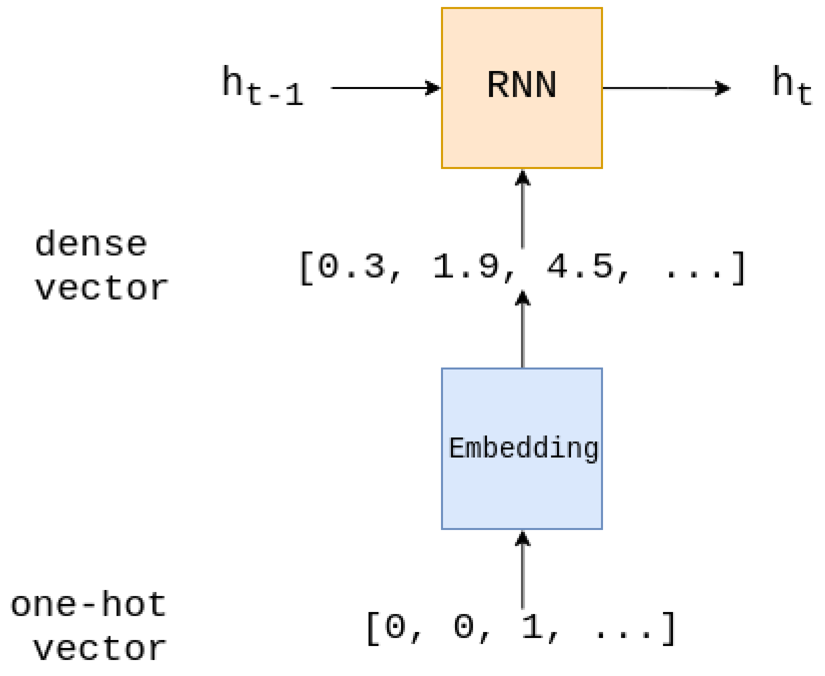




Sentiment Analysis

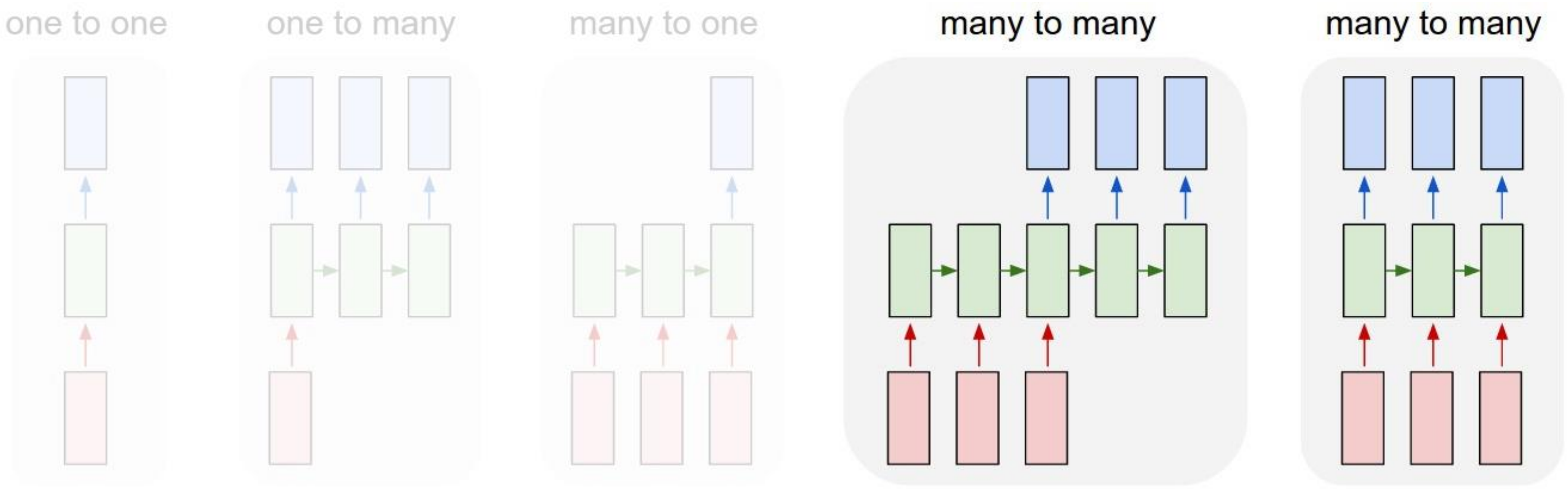
Many-to-one: Simple sentiment analysis

```
import torch.nn as nn

class RNN(nn.Module):
    def __init__(self, input_dim, embedding_dim, hidden_dim, output_dim):

        super().__init__()

        self.embedding = nn.Embedding(input_dim, embedding_dim)

        self.rnn = nn.RNN(embedding_dim, hidden_dim)

        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, text):

        #text = [sent len, batch size]

        embedded = self.embedding(text)

        #embedded = [sent len, batch size, emb dim]

        output, hidden = self.rnn(embedded)

        #output = [sent len, batch size, hid dim]
        #hidden = [1, batch size, hid dim]

        assert torch.equal(output[-1,:,:], hidden.squeeze(0))

        return self.fc(hidden.squeeze(0))
```

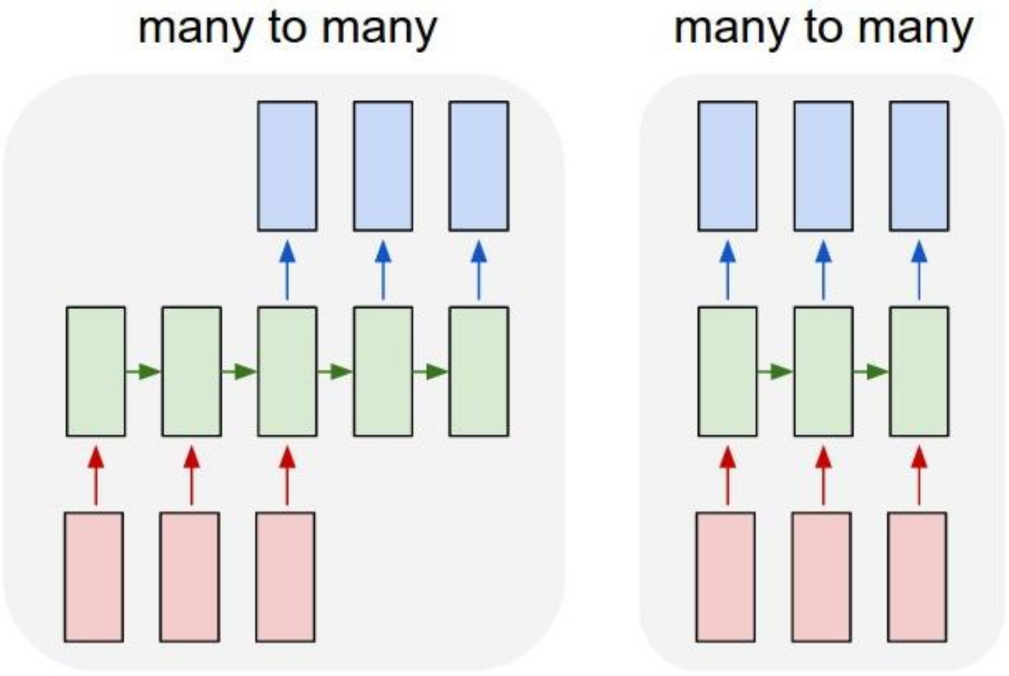


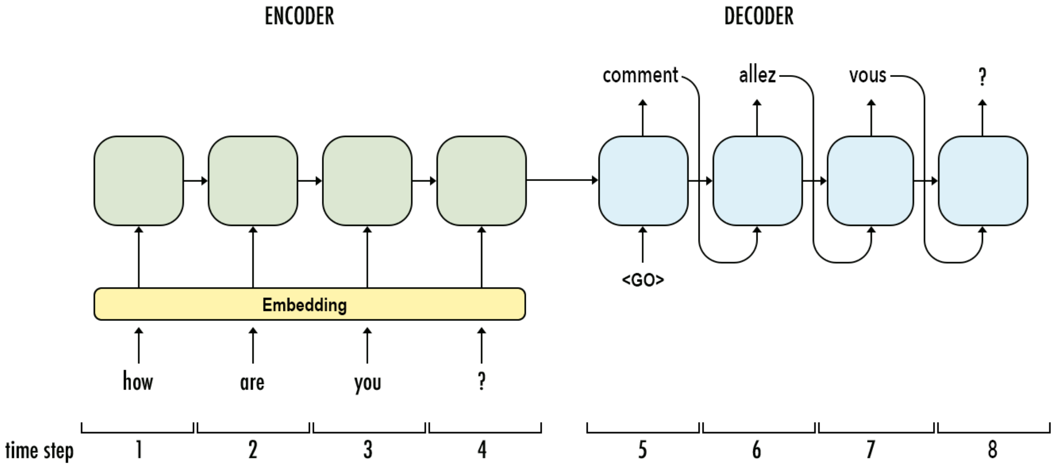
Implementation detail : https://towardsdatascience.com/sentiment-analysis-using-lstm-step-by-step-50d074f09948

Many-to-Many

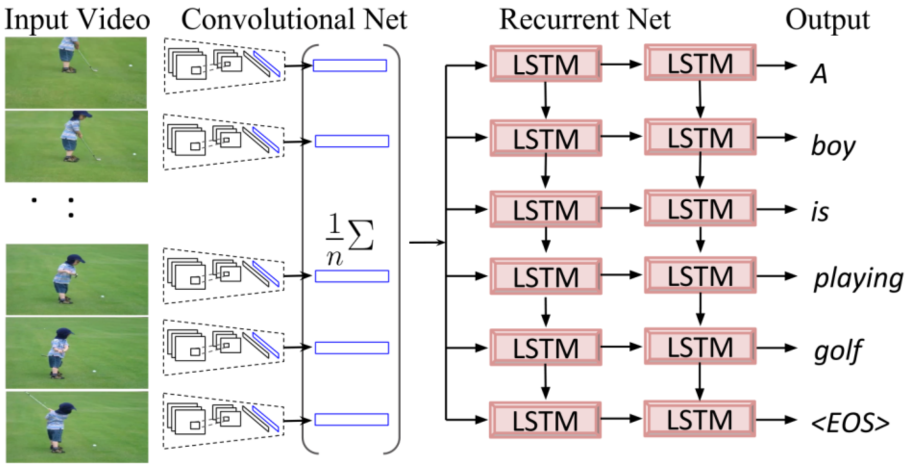


Many-to-Many





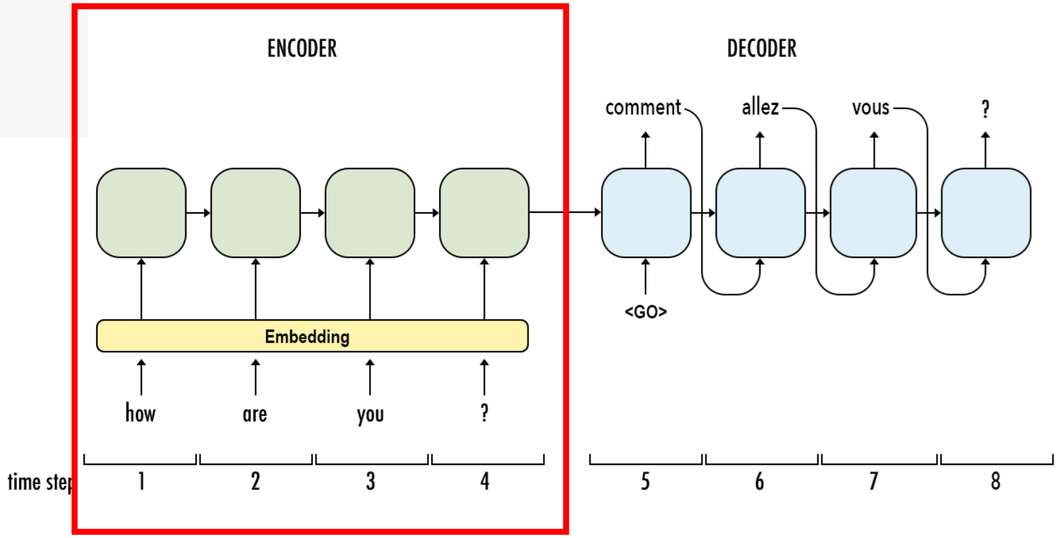
Machine Translation



Video Captioning

Many-to-Many: Machine translation example

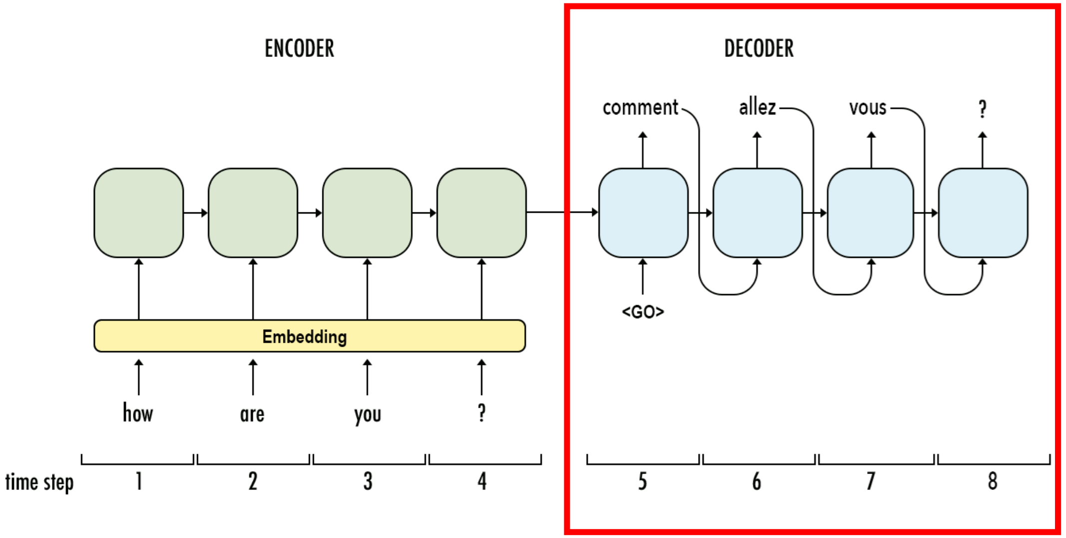
```
class Encoder(nn.Module):
    def __init__(self, vocab_len, embedding_dim, hidden_dim, n_layers, dropout_prob):
        super().__init__()

        self.embedding = nn.Embedding(vocab_len, embedding_dim)
        self.rnn = nn.LSTM(embedding_dim, hidden_dim, n_layers, dropout=dropout_prob)

        self.dropout = nn.Dropout(dropout_prob)

    def forward(self, input_batch):
        embed = self.dropout(self.embedding(input_batch))
        outputs, (hidden, cell) = self.rnn(embed)

        return hidden, cell
```



Many-to-Many: Machine translation example

```
class OneStepDecoder(nn.Module):
    def __init__(self, input_output_dim, embedding_dim, hidden_dim, n_layers, dropout_prob):
        super().__init__()
        # self.input_output_dim will be used later
        self.input_output_dim = input_output_dim

        self.embedding = nn.Embedding(input_output_dim, embedding_dim)
        self.rnn = nn.LSTM(embedding_dim, hidden_dim, n_layers, dropout=dropout_prob)
        self.fc = nn.Linear(hidden_dim, input_output_dim)
        self.dropout = nn.Dropout(dropout_prob)

    def forward(self, target_token, hidden, cell):
        target_token = target_token.unsqueeze(0)
        embedding_layer = self.dropout(self.embedding(target_token))
        output, (hidden, cell) = self.rnn(embedding_layer, (hidden, cell))

        linear = self.fc(output.squeeze(0))

        return linear, hidden, cell
```

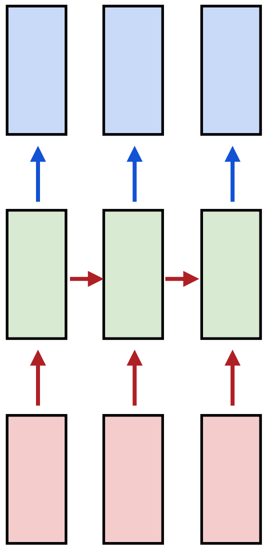
```
class Decoder(nn.Module):
    def __init__(self, one_step_decoder, device):
        super().__init__()
        self.one_step_decoder = one_step_decoder
        self.device=device

    def forward(self, target, hidden, cell):
        target_len, batch_size = target.shape[0], target.shape[1]
        target_vocab_size = self.one_step_decoder.input_output_dim
        # Store the predictions in an array for loss calculations
        predictions = torch.zeros(target_len, batch_size, target_vocab_size).to(self.device)
        # Take the very first word token, which will be sos
        input = target[0, :]

        # Loop through all the time steps
        for t in range(target_len):
            predict, hidden, cell = self.one_step_decoder(input, hidden, cell)

            predictions[t] = predict
            input= predict.argmax(1)

        return outputs
```



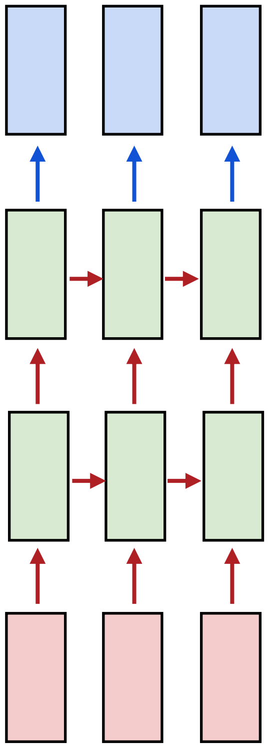
Implementation detail :
http://www.adeveloperdiary.com/data-science/deep-learning/nlp/machine-translation-recurrent-neural-network-pytorch/

Part 3: Other RNN Variants

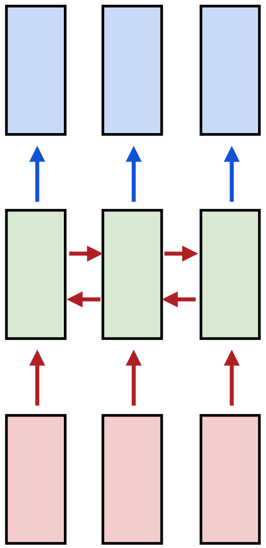
Deep RNN and Bi-directional RNN



Regular RNN

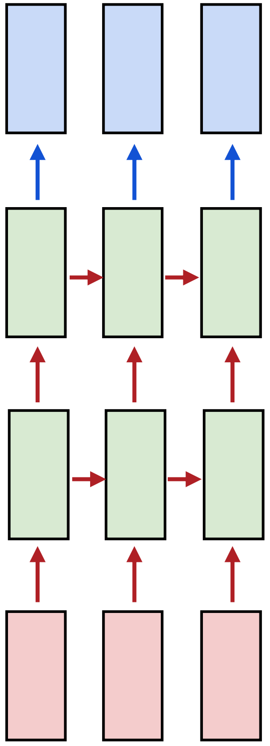


Deep RNN



Bi-directional RNN

DRNN: Pros and Cons



Deep RNN

(+)

Can provide better performance

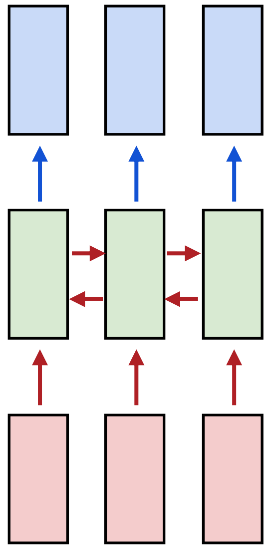
Often used for complex problems

(-)

Potential for overfitting

Longer training time

Bi-RNN: Pros and Cons



Bi-directional RNN

(+)
Higher performance in NLP tasks
Suitable when both left and right contexts are used

(-)
Harder to train than Uni-directional RNN
Not suitable for real-time processing

DRNN/Bi-RNN Implementation

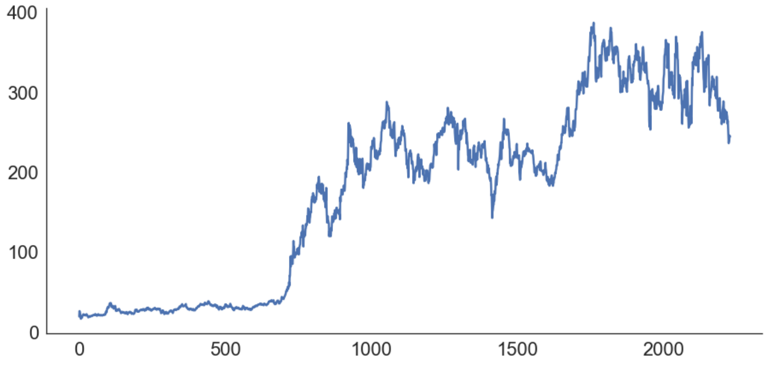
```
self.rnn = nn.LSTM(embedding_dim, hidden_dim, n_layers = 2, dropout=dropout_prob, bidirectional = True)
```

Number of recurrent layers

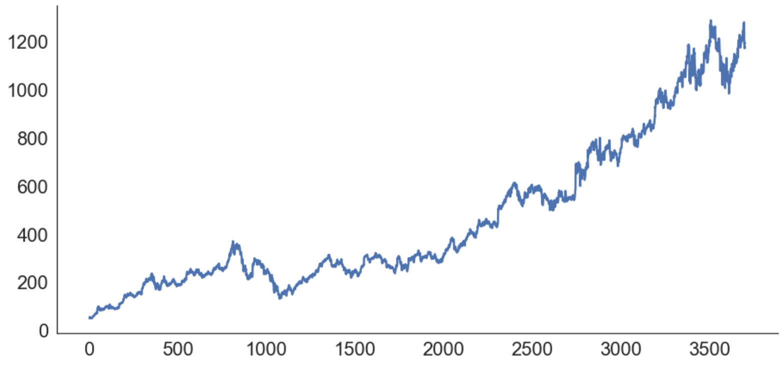
Enable bi-directionality

Lab Assignment: Predict stock prices using RNNs

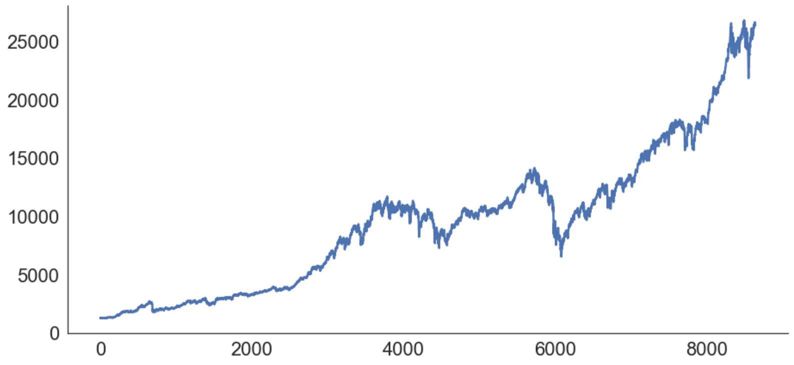
Dataset



Tesla



Google



Dow Jones

Task: Use many-to-one RNN architecture of choice to predict 100 days of stock values.

Evaluation: Plot the ground truth and predicted values for the last 100 days.