Outline

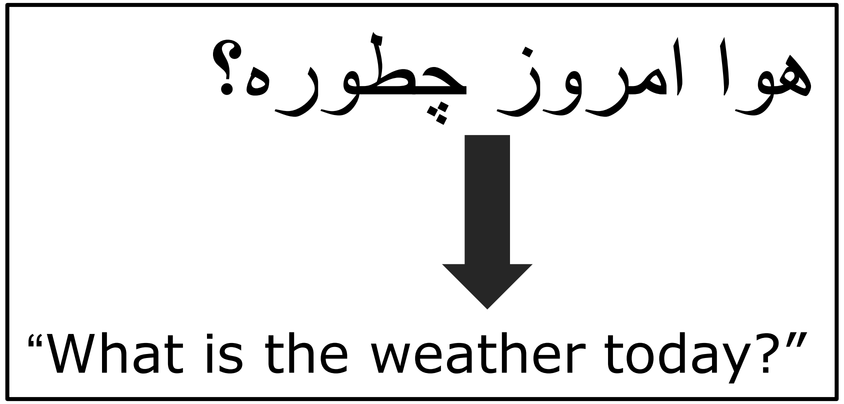
- Sequential Data
- Intro to Recurrent Neural Networks (RNNs)
- Challenges with RNNs
- Example: Sine Wave Generation
- Assignment: Cosine Waves

Sequential Data

Example Sequential Data and Tasks

- Written Language
  - Character Prediction
  - Machine Translation
- Audio
  - Speech Processing
  - Music Transcription
- Spatio-Temporal Data
  - Body capture data
  - Weather modelling
  - Neurological Models

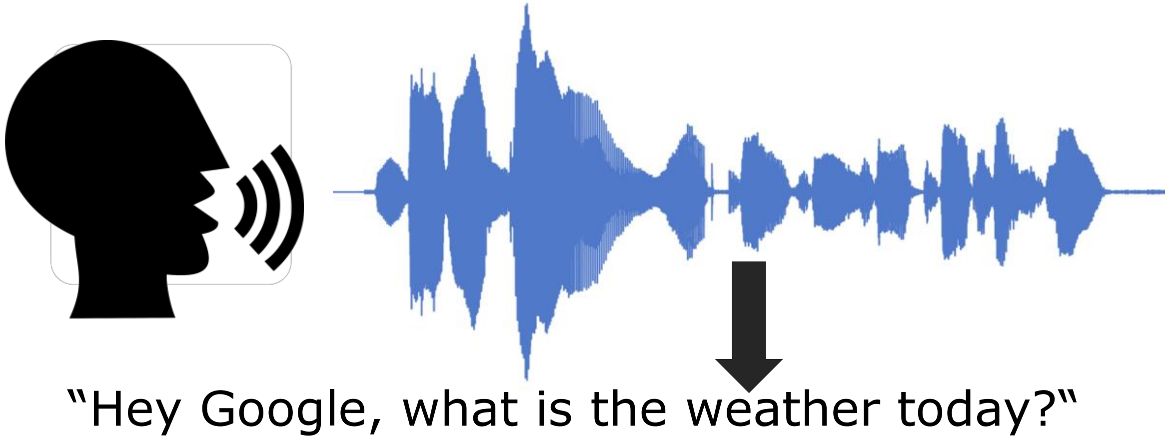
Language Modelling/Prediction



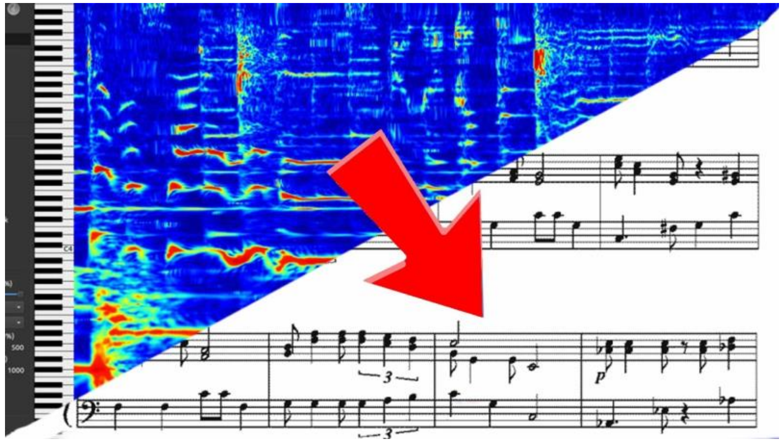
Machine Translation

Example Sequential Data and Tasks

- Written Language
  - Character Prediction
  - Machine Translation
- Audio
  - Speech Recognition
  - Music Transcription
- Spatio-Temporal Data
  - Body capture data
  - Weather modelling
  - Neurological Models



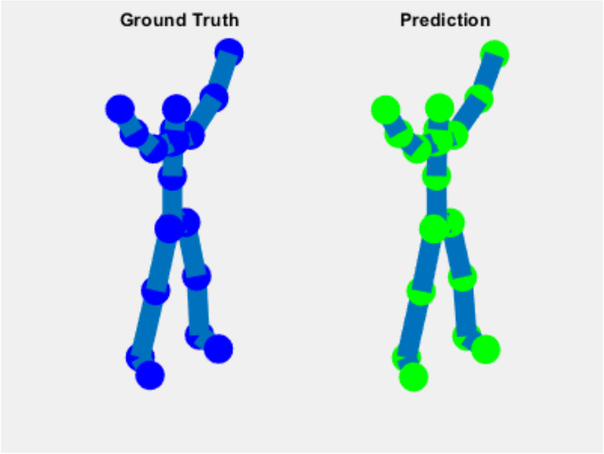
Speech Recognition



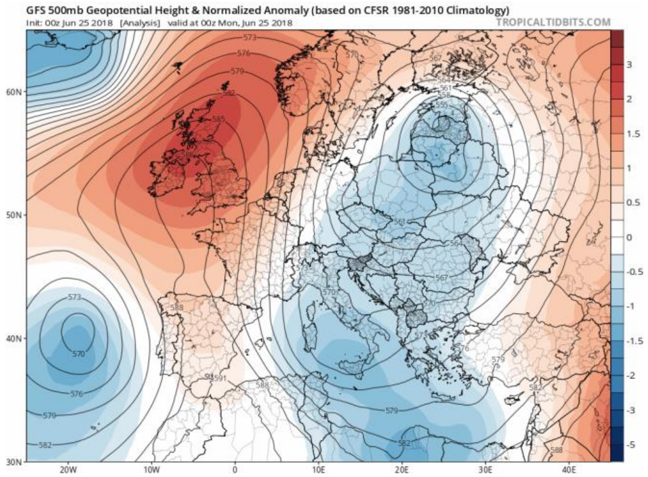
Music Transcription

Example Sequential Data and Tasks

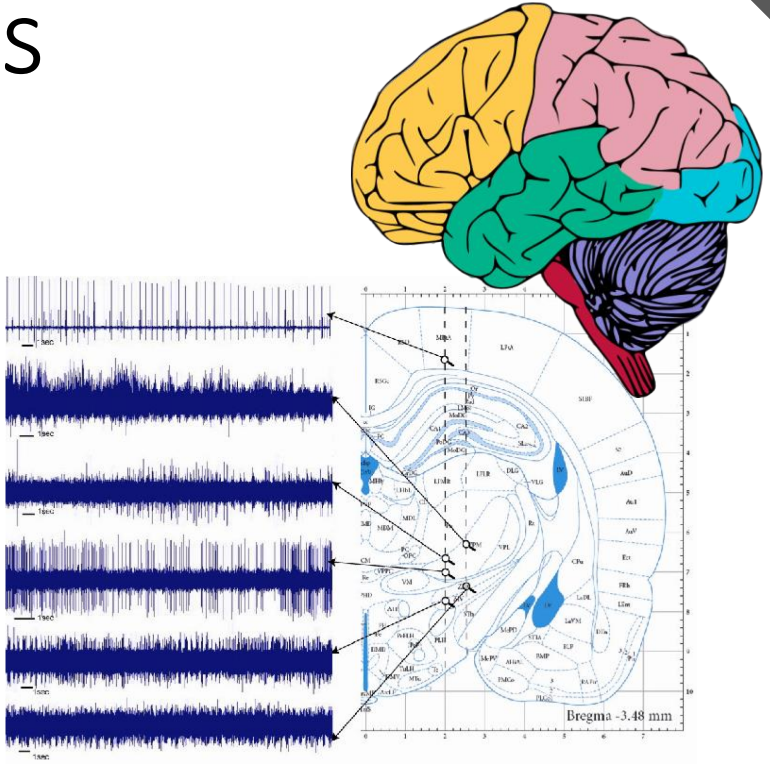
- Written Language
  - Character Prediction
  - Machine Translation
- Audio
  - Speech Recognition
  - Music Transcription
- Spatio-Temporal Data
  - Body capture data
  - Weather modelling
  - Neurological Data



Body Motion Capture



Weather Modelling



Brain Activity

Features of Sequential Data

- Order matters
  - Unlike a group of vectors, the order in which data appear in sequential data is important
- Variable length
  - Measurements are not always captured over the same number of time-steps
- Temporal dependence
  - Previous data values usually has impact on current value

Recurrent Neural Networks

RNN Setup

- Processes input (x) at each step using shared parameters (A)
- Retains memory by evolving the hidden state (h) as a function of input

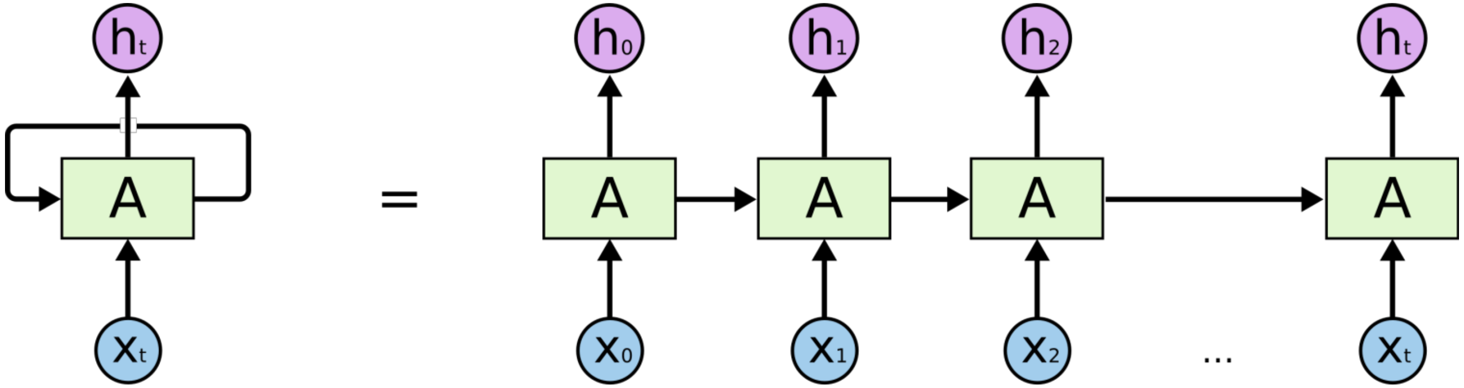


Image Source: [Colah's Blog]

RNN Setup

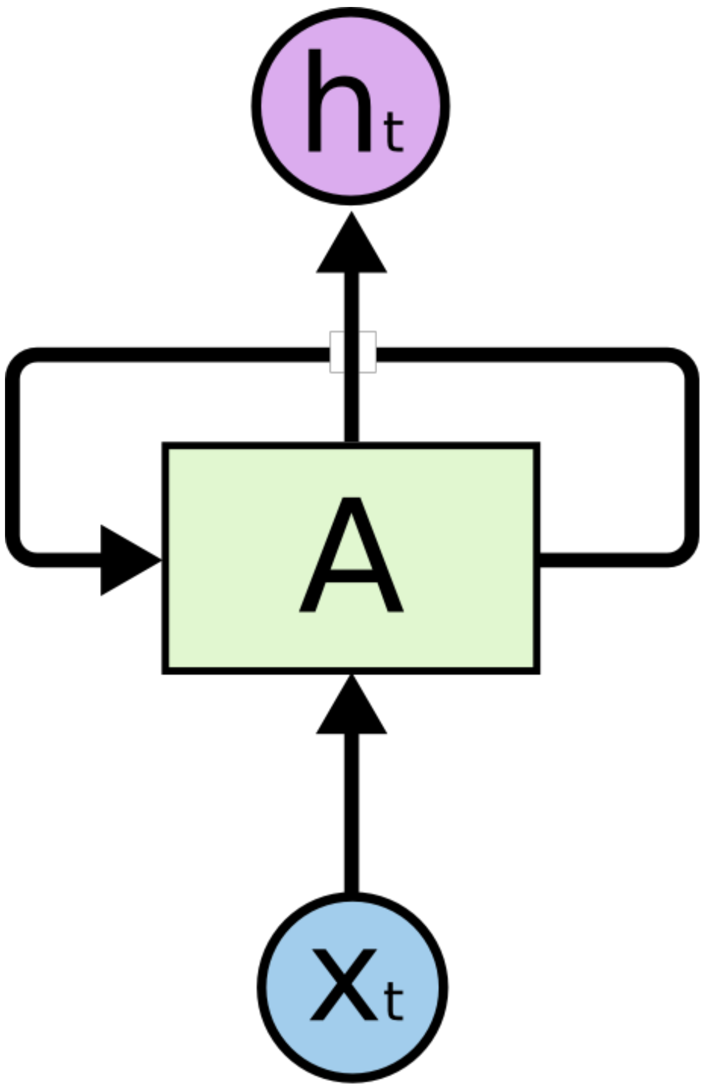
- Output of a cell is the hidden state $h^{(t)}$
- Function of input (x) at time t and previous hidden state

$$
\begin{aligned}
\boldsymbol{a}^{(t)} &= \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)} \\
\boldsymbol{h}^{(t)} &= \tanh(\boldsymbol{a}^{(t)})
\end{aligned}
$$

RNN Implementation

`torch.nn.RNN()`

- Parameters
  - input_size
  - hidden_size
  - num_layers
  - nonlinearity
  - bias
  - batch_first
  - dropout
  - bidirectional



RNN Implementation

`torch.nn.RNN()`

- Parameters
  - input_size
  - hidden_size
  - num_layers
  - nonlinearity
  - bias
  - batch_first
  - dropout
  - bidirectional

- Input size is the dimension of the input at each time step.
  - The size of each vector in a sequence
- Hidden size is the dimensionality of the hidden state, h.
  - The number of neurons processing each time step

RNN Implementation

`torch.nn.RNN()`

- Parameters
  - input_size
  - hidden_size
  - num_layers
  - nonlinearity
  - bias
  - batch_first
  - dropout
  - bidirectional

- Num_layers is the number of stacked RNN layers (default: 1)
  - Will be covered in detail in Lab 6
- Non-linearity is the activation function for the neurons
  - Choice of `tanh` or `relu` (default: `tanh`)

RNN Implementation

`torch.nn.RNN()`

- Parameters
  - input_size
  - hidden_size
  - num_layers
  - nonlinearity
  - bias
  - batch_first
  - dropout
  - bidirectional

- Bias is a Boolean indicating whether to include the bias term (b) in the neuron (default: True)

$$a^{(t)} = b + Wh^{(t-1)} + Ux^{(t)}$$

- Batch_first indicates whether the first dimension of the input is the batch- *(batch, seq, feature)*
  - Default: False – *(seq, batch, feature)*

RNN Implementation

`torch.nn.RNN()`

- Parameters
  - input_size
  - hidden_size
  - num_layers
  - nonlinearity
  - bias
  - batch_first
  - dropout
  - bidirectional

- Dropout: if non-zero, introduces a *Dropout* layer on the outputs of each RNN layer except the last
  - No effect if num_layers = 1
- Bidirectional: Boolean indicating whether the RNN is bidirectional (Default: False)

RNN Usage

- Inputs:
  - `input` is a tensor containing the features of the input sequence
    - Default shape: *(seq_len, batch, input_size)* – can change if batch_first
    - Can be packed variable length sequence (see torch.nn.utils.rnn.pack_padded_sequence() or torch.nn.utils.rnn.pack_sequence())

RNN Usage

- Inputs:
  - `input` is a tensor containing the features of the input sequence
    - Default shape: *(seq_len, batch, input_size)* – can change if batch_first
    - Can be packed variable length sequence (see [torch.nn.utils.rnn.pack_padded_sequence()]() or [torch.nn.utils.rnn.pack_sequence()]())
  - `h_0` is a tensor containing the initial hidden state for each element in batch
    - Shape: *(num_layers*num_directions, batch, hidden_size)*
    - Common practice is to use `torch.zeros(num_layers*num_directions, batch, hidden_size)`
    - If None, will default to torch.zeros of the appropriate size

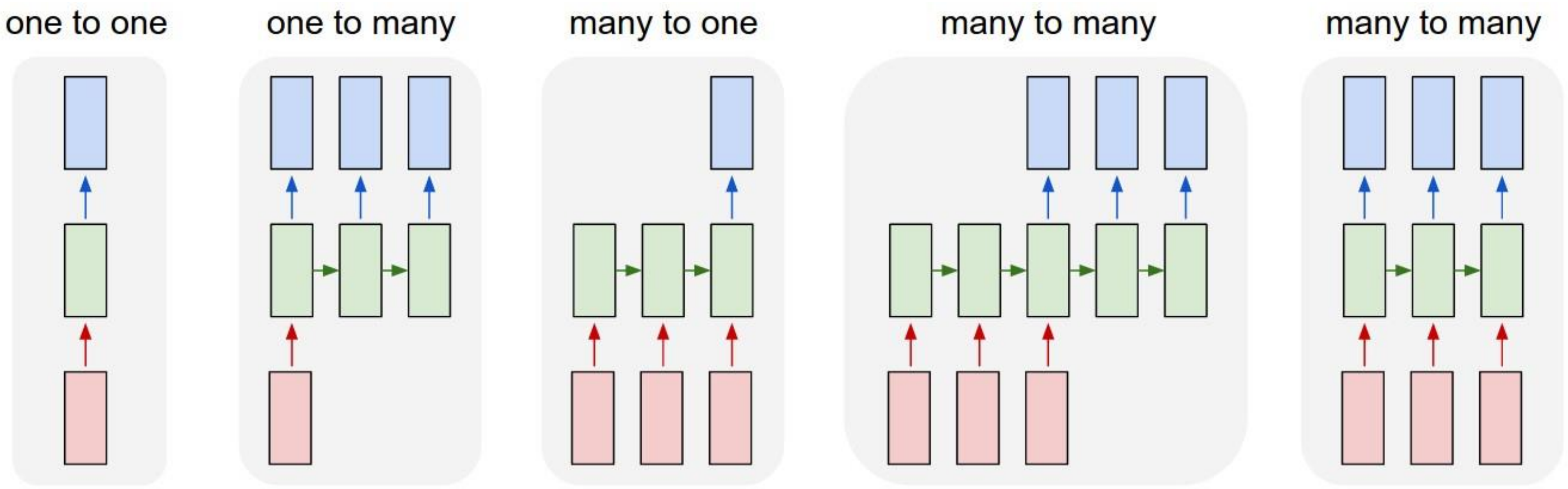
RNN Usage

- Outputs
  - `output` is a tensor containing the output features *(h_t)* from the **last** layer of the RNN, for each *t*.
    - Default shape: *(seq_len, batch, num_directions*hidden_size)*

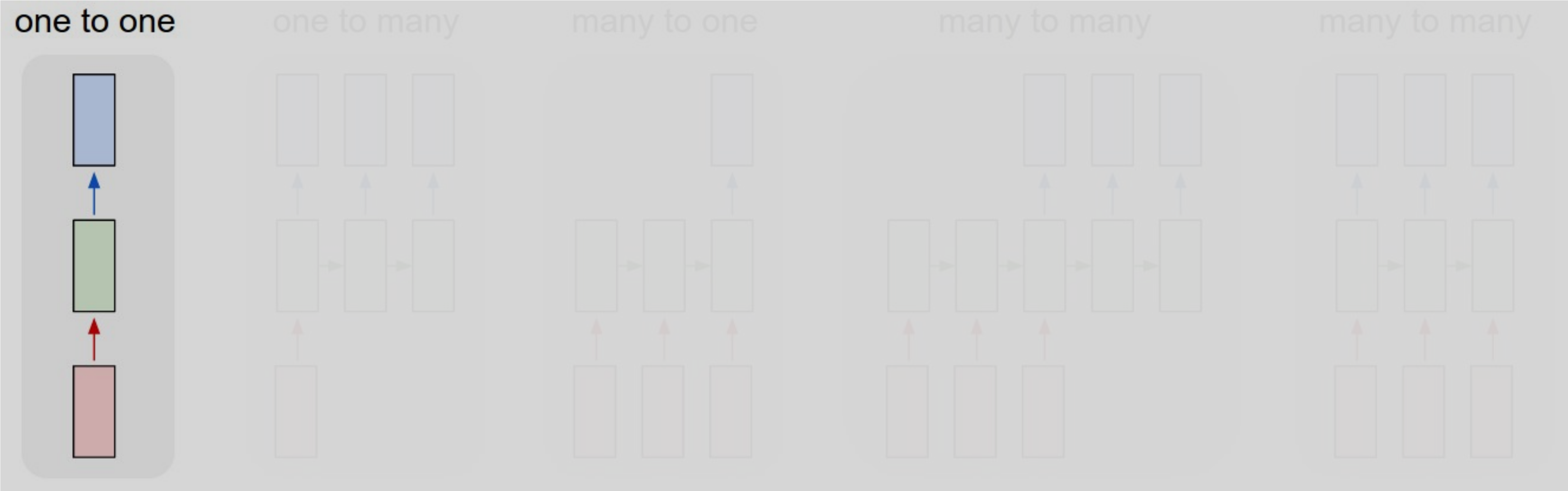
RNN Usage

- Outputs
  - `output` is a tensor containing the output features *(h_t)* from the **last** layer of the RNN, for each *t*.
    - Default shape: *(seq_len, batch, num_directions*hidden_size)*
  - h_n is a tensor containing the all the hidden states for t = seq_len (i.e., after the final time step of the sequence)
    - Shape: *(num_layers*num_directions, batch, hidden_size)*

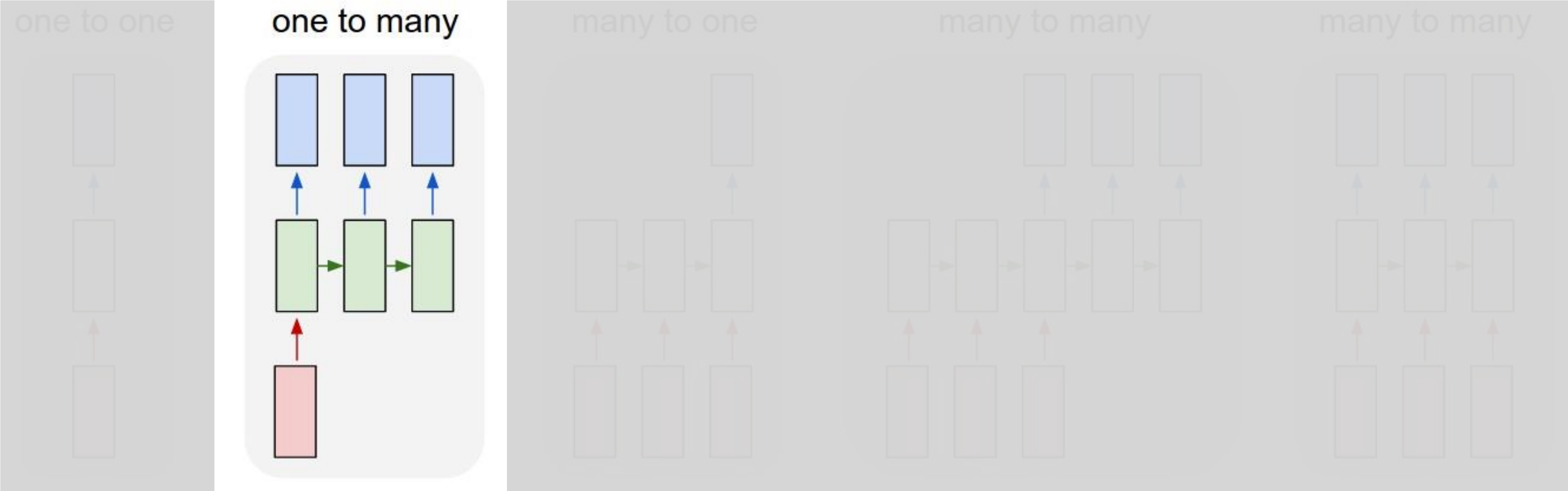
RNN Problem Types



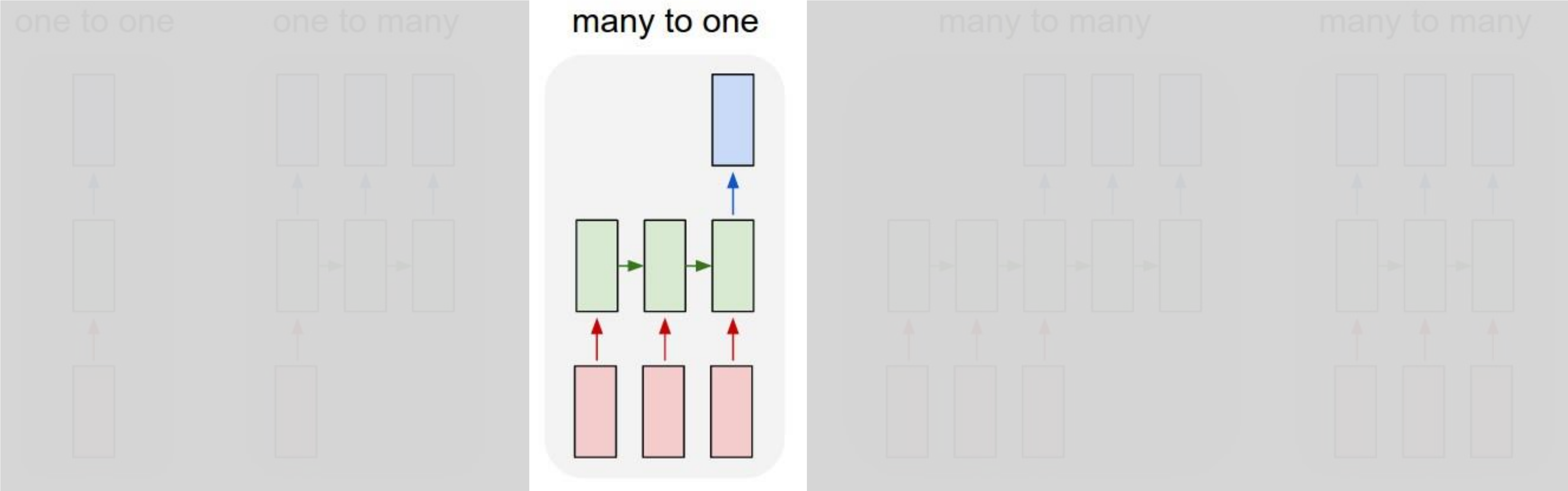
RNN Problem Types



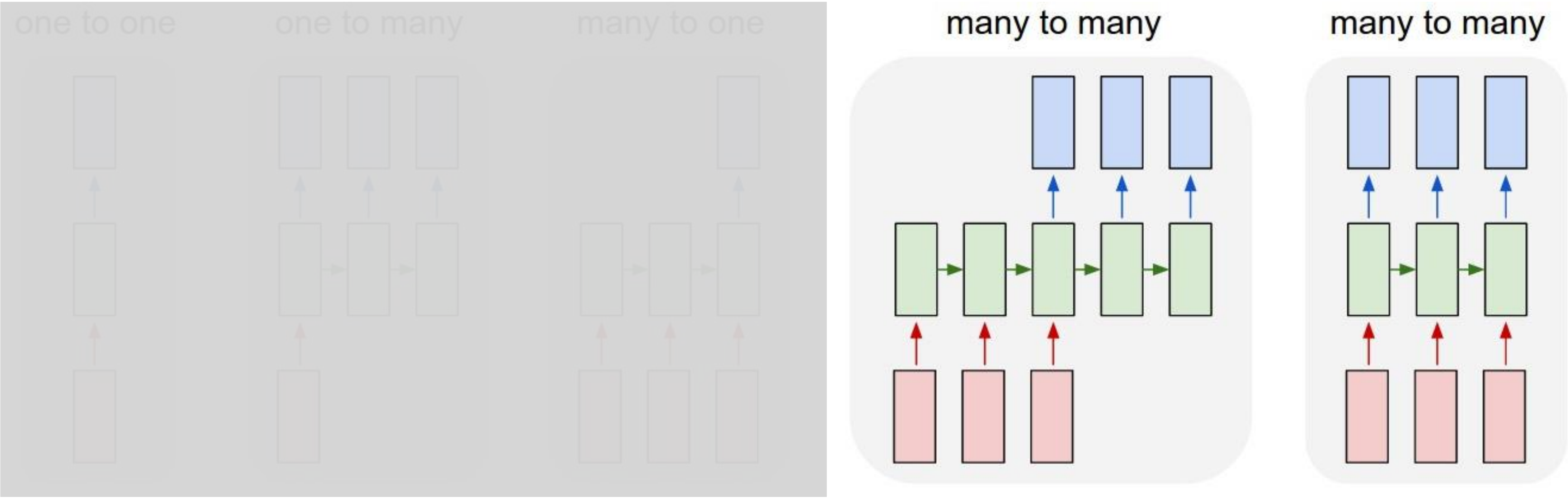
RNN Problem Types



RNN Problem Types



RNN Problem Types



RNN Challenges

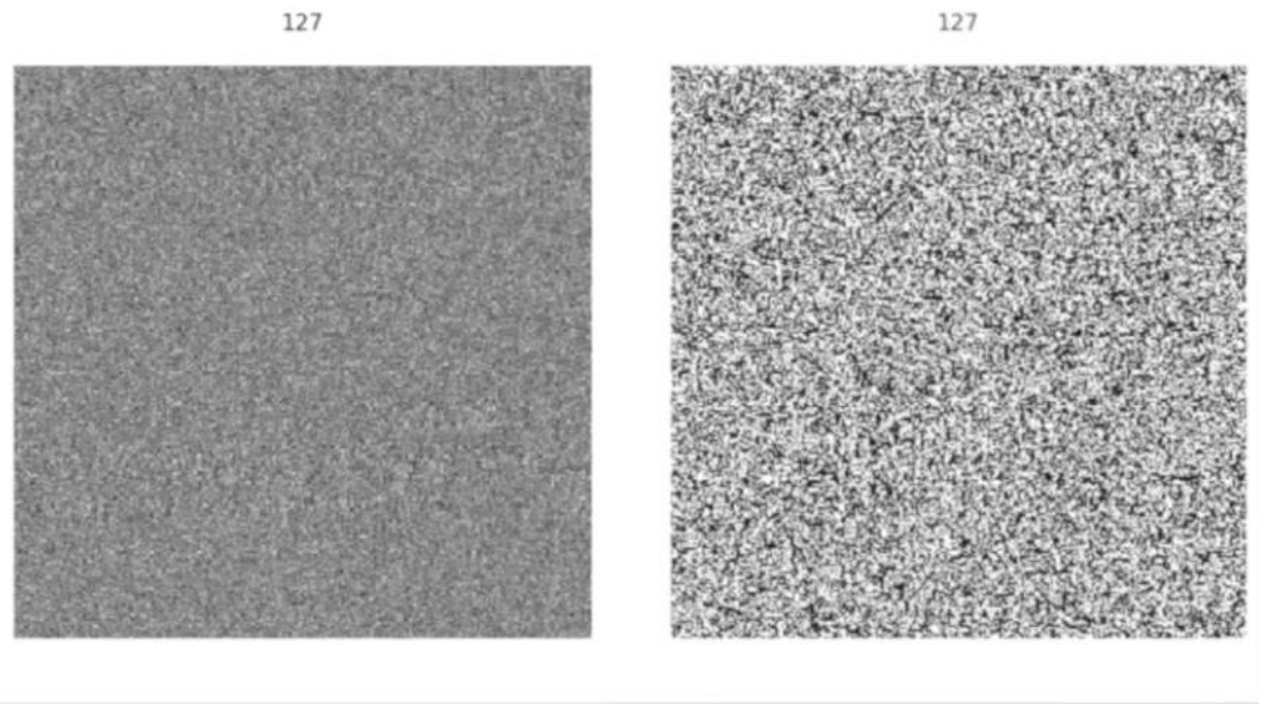
Vanishing and Exploding Gradient

- To learn on sequential data, we need:
  - Sensitivity to new input
  - Retention of old information

- Backpropagation over large number of steps leads to issues
  - Amplification of patterns by repeated application of same function

Vanishing and Exploding Gradient

- Long-term memory of RNNs suffers due to uncontrolled gradients
- Additional mechanisms are used to optimize memory and sensitivity (details in Lab 6)
  - Gated architectures – LSTM, GRU
  - Constrained-weight RNNs



Sensitivity to Initialization

- RNN performance can be very sensitive to initialization
- Optimal initialization distribution depends on:
  - Hidden size
  - Non-linearity
  - Input statistics

Initialization Distributions in PyTorch

torch.nn.init

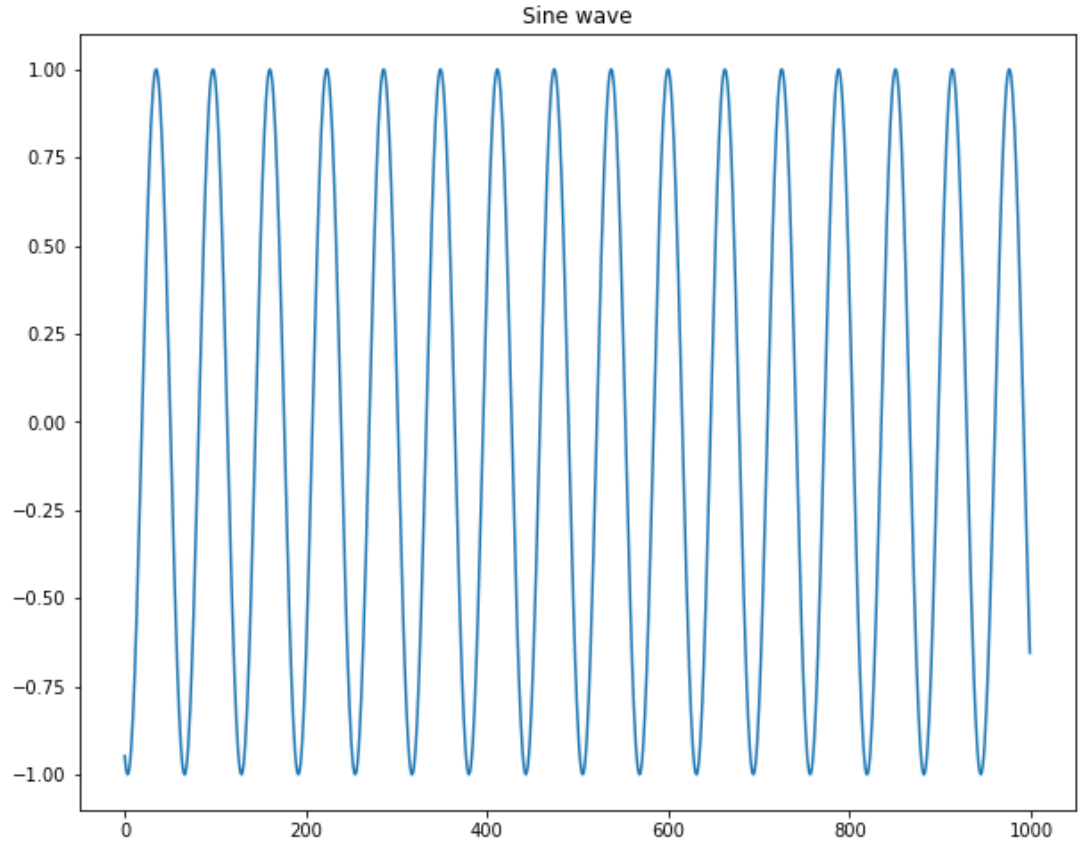
- Uniform
- Normal
- Xavier Uniform
- Xavier Normal
- Kaiming Uniform
- Kaiming Normal
- Orthogonal

Example: Sine Wave Generation

Adapted this tutorial: https://lirnli.wordpress.com/2017/09/01/simple-pytorch-rnn-examples/

Problem setup

- Sine wave with random shift as input
- Want to predict the next time step given an input sequence



Model Definition

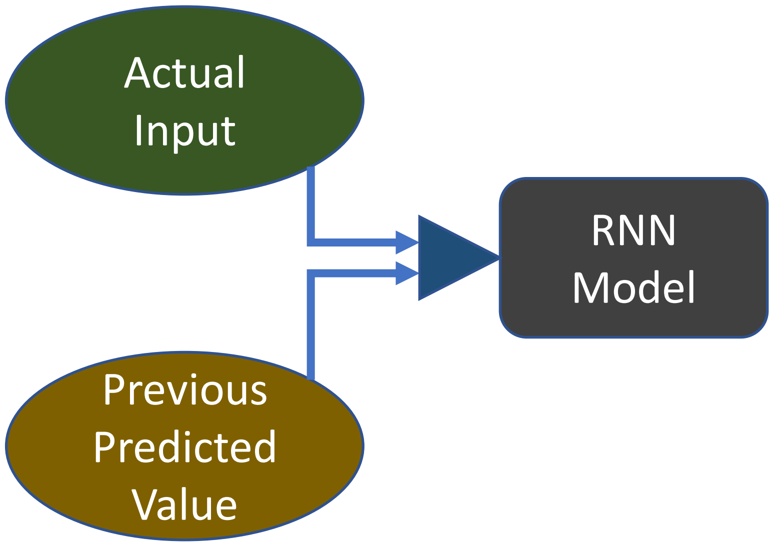
- Using RNN with hidden size 128
- Use linear layer for output
- Teacher forcing to help guide network with true values
  - hybrid one-to-many/many-to-many problem

```
# Model Definition
class SineRNN(nn.Module):
  def __init__(self, p = 0.5):
    super(SineRNN, self).__init__()
    self.rnn_layer = nn.RNN(input_size = 1, hidden_size = 128)
    self.out_layer = nn.Linear(in_features = 128, out_features = 1)
    self.p = p #Whether to use actual seq or output for next step

  def forward(self,seq, h = None):
        out = []
        X_in = torch.unsqueeze(seq[0],0)
        for X in seq:
            if np.random.rand()>self.p: #Use teacher forcing
                X_in = X.unsqueeze(dim = 0)
            tmp, h1 = self.rnn_layer(X_in, h1)
            X_in = self.out_layer(tmp)
            out.append(X_in)
        return torch.stack(out).squeeze(1), h1
```

Teacher Forcing

- When predicting a sequence from an input sequence, you can use "teacher forcing"
- Randomly select either:
  - Actual input (teacher) value
  - Output of network at previous step (prediction)



Teacher Forcing

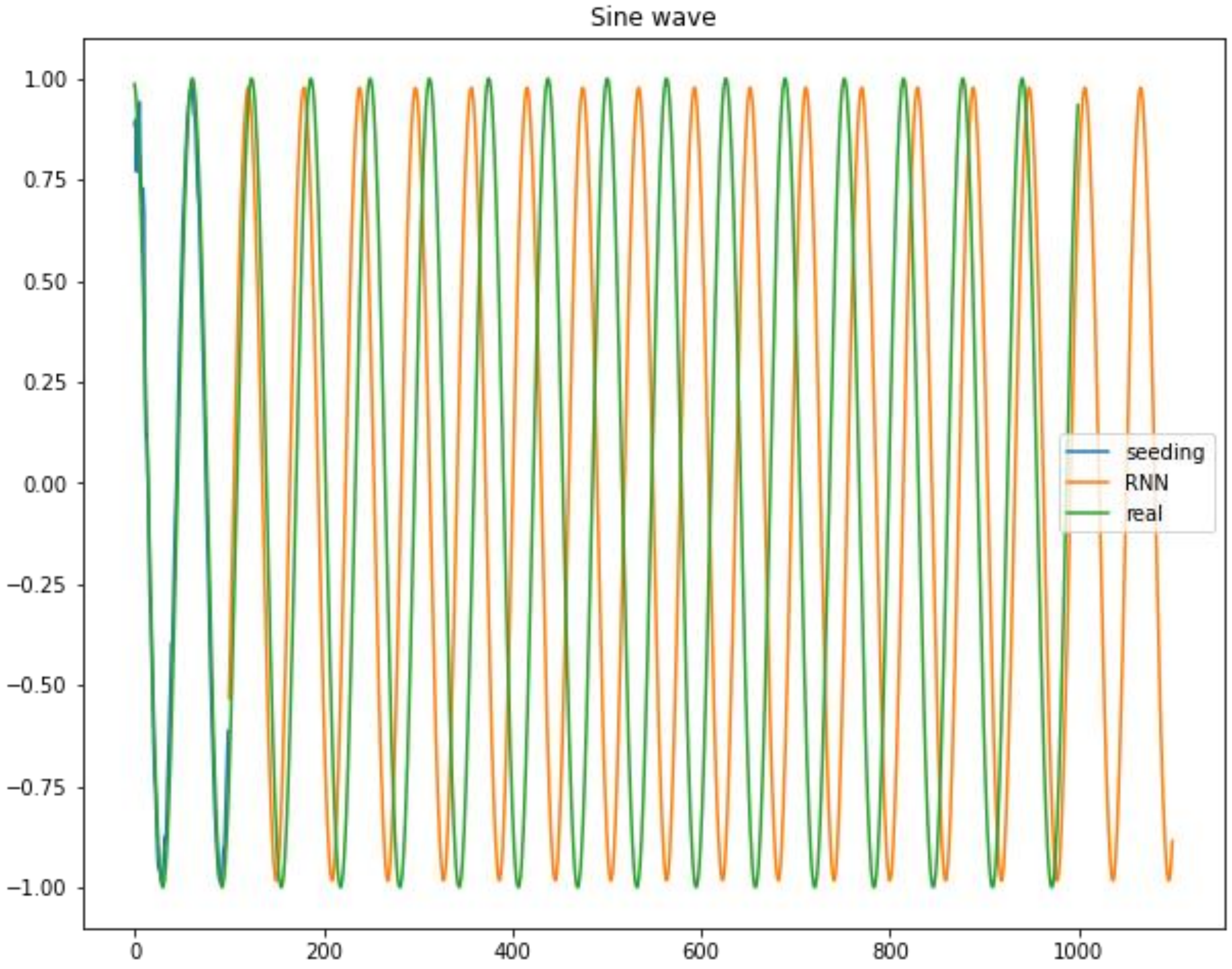
Training

- Generate data:
  - Sine waves with random shifts
  - Both input and targets (shifted 1)
- Increase chance of using past predictions
- Warm-up for 20 steps and then evaluate loss

```
seq = SineRNN()
criterion = nn.MSELoss()
optimizer = optim.Adam(seq.parameters(), lr=0.001)
max_iters = 10000
train_loss = []
for i in range(max_iters):
    data = np.sin(np.linspace(0,10,100)+2*np.pi*np.random.rand())
    xs = data[:-1]
    ys = data[1:]
    X = torch.Tensor(xs).view(-1,1,1)
    y = torch.Tensor(ys)
    if i%100==0:
        seq.p = min(seq.p+0.1,0.85)  # encourage training longer term predictions
    optimizer.zero_grad()
    rnn_out,_ = seq(X)
    loss = criterion(rnn_out[20:].view(-1),y[20:])
    loss.backward()
    optimizer.step()
    train_loss.append(loss.item())
    if i%500 == 0:
        print(f"i {i}, loss {loss.data:.4f}")
```

Results- Predictions

- First 100 steps as seed
- Predict for 1000 steps
- Able to capture general shape, but makes errors finding period (requires long memory)

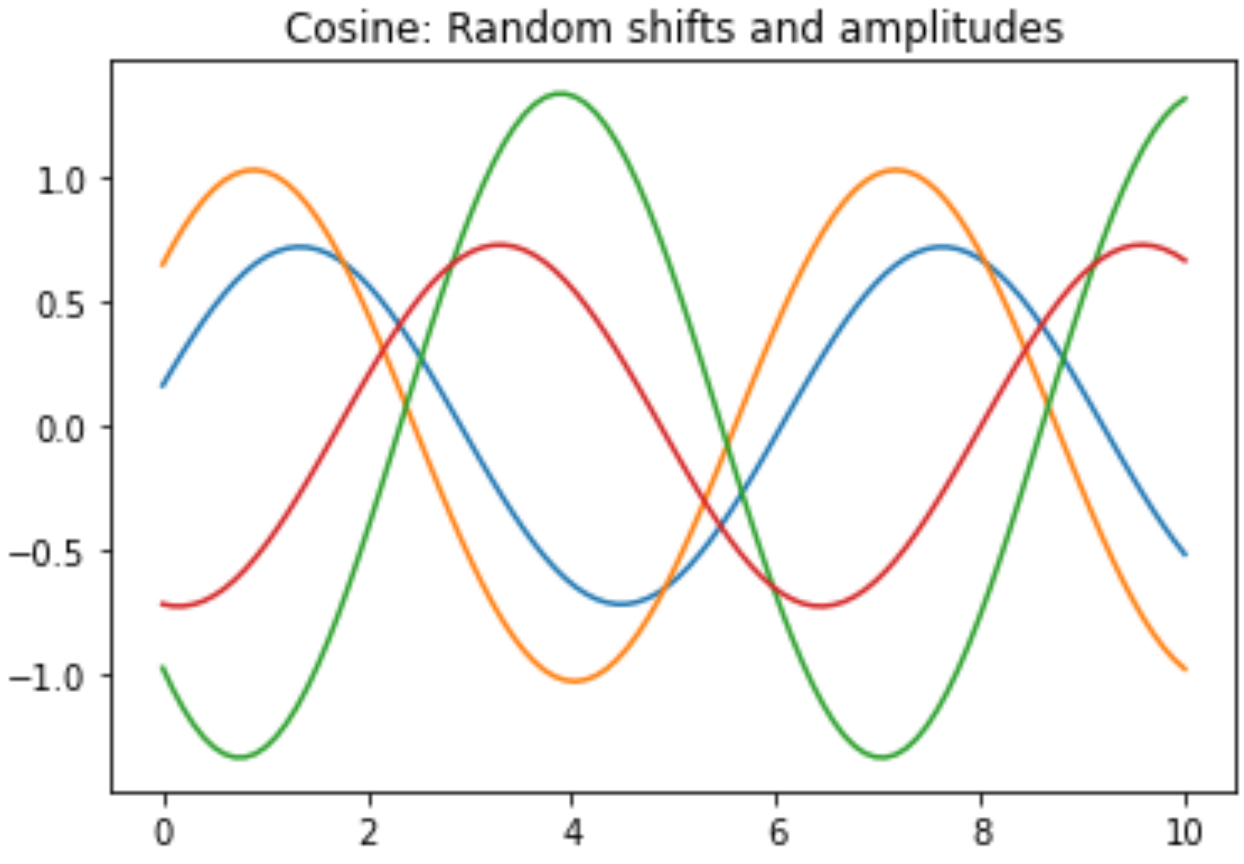


Total MSE Loss: 0.9695

Assignment: Cosine Wave Generation

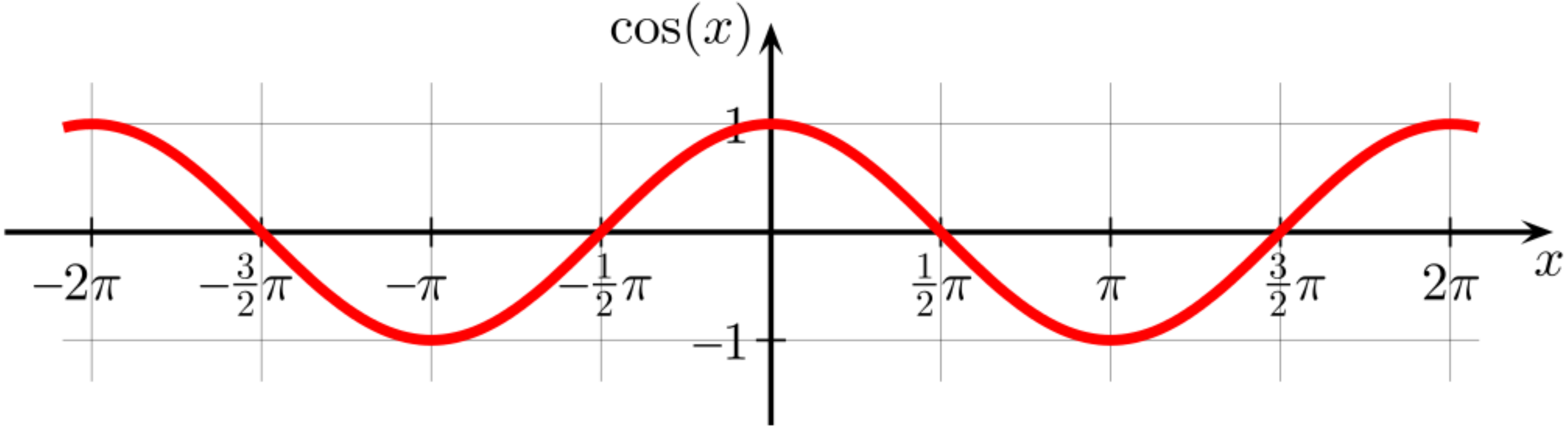
Assignment Details

- Create an RNN model to generate cosines like in example
- Use random shifts as in example
- Generate samples with different amplitudes, ranging from 0.5 to 1.5 (How much data do you need?)
- Play with hyperparameters of your network



Evaluation

- Generate a validation/test set of 1000 randomly-generated cosines
- Find MSE loss on this validation set
- Plot your best and worst predictions in the validation set



Evaluation Detail

- For validation set, create longer sequences (1000 steps)
- Evaluate loss after a warm-up period of 100 steps
- Only calculate loss where prediction and real input overlap