LAB 6:
ADVANCED RECURRENT NERUAL NETWORKS

University of Washington, Seattle

Fall 2024

OUTLINE

Part 1: Gated RNNs

- Need for Gated RNNs
- LSTM
- GRU

Part 2: Training Gated RNNs

- Mini-batch Gradient in RNNs
- RNN extensions on LSTM/GRU

Part 3: Gated RNN Application with PyTorch

- Signal Denoising

Part 4: Encoder-Decoder RNNs in PyTorch

- Signal Prediction

Part 5: Lab Assignment

- Stock Prediction

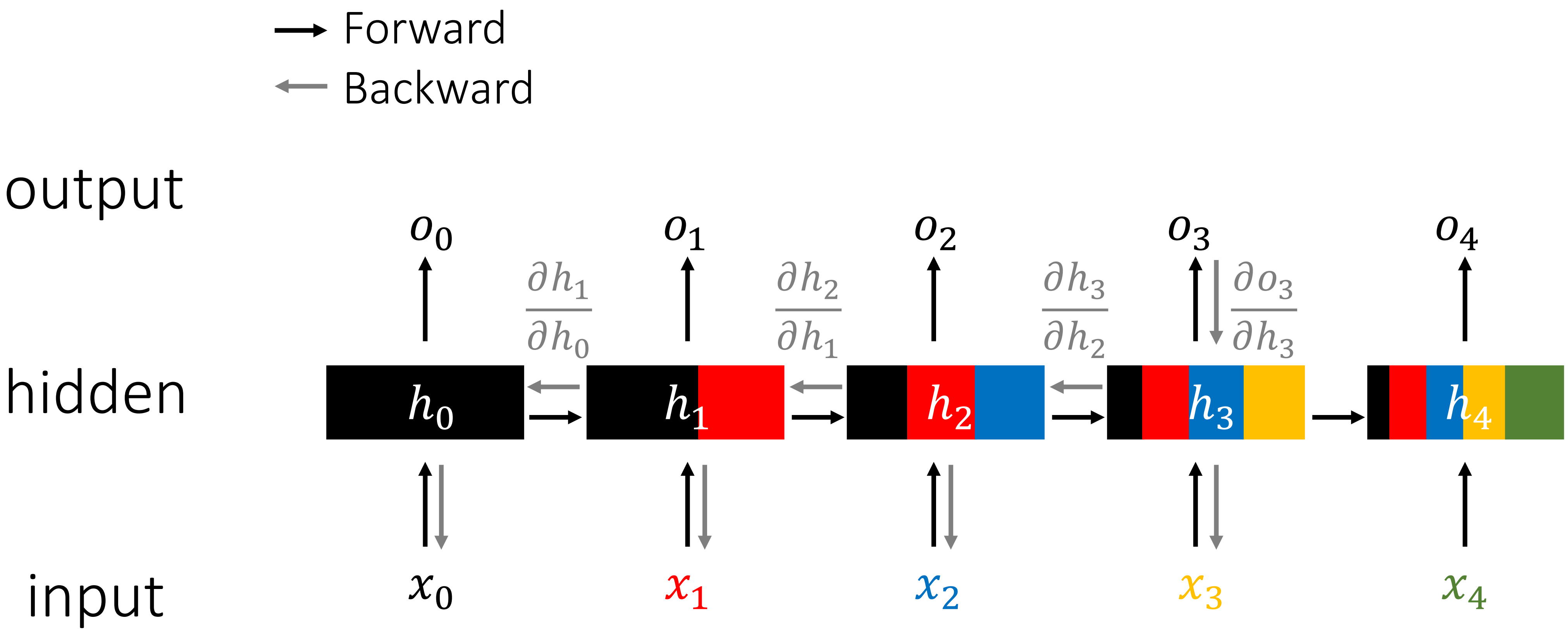
GATED RNNs

Need for Gated RNNs

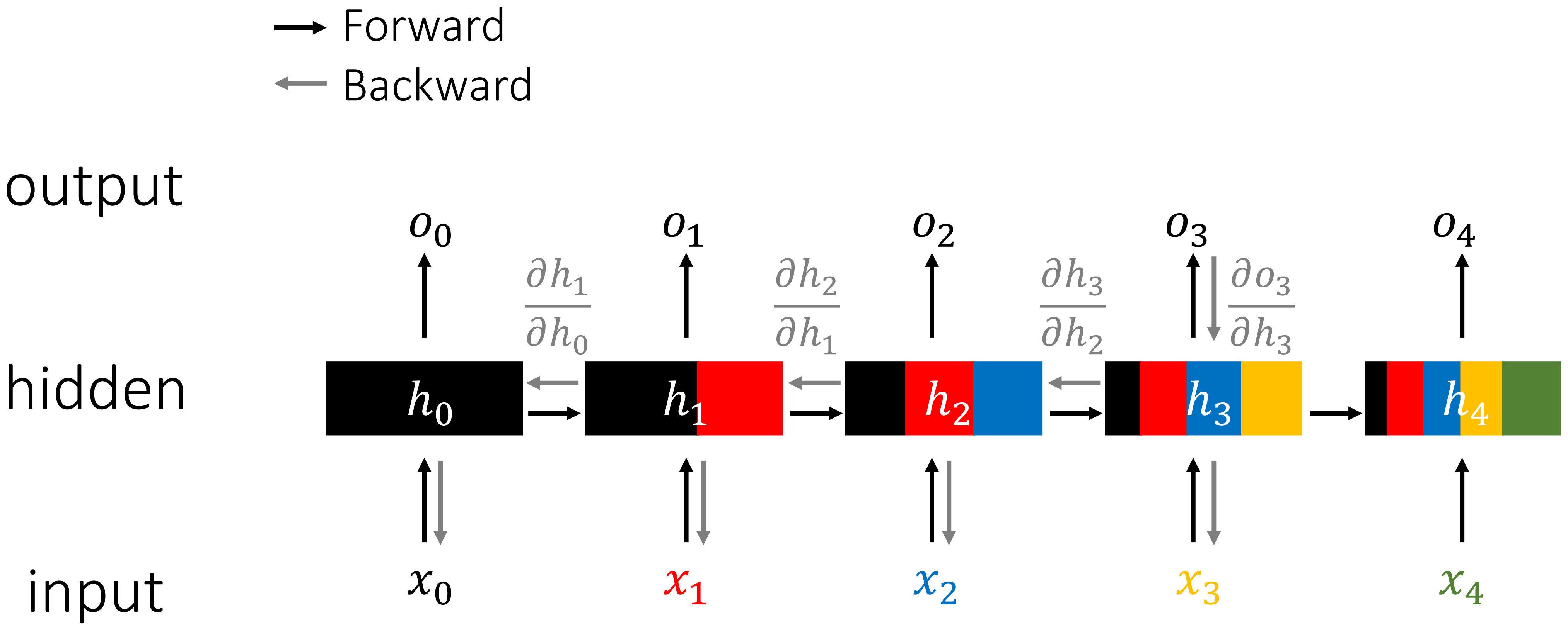
Long Short-Term Memory (LSTM)

Gated Recurrent Unit (GRU)

Recap: Backpropagation in RNNs

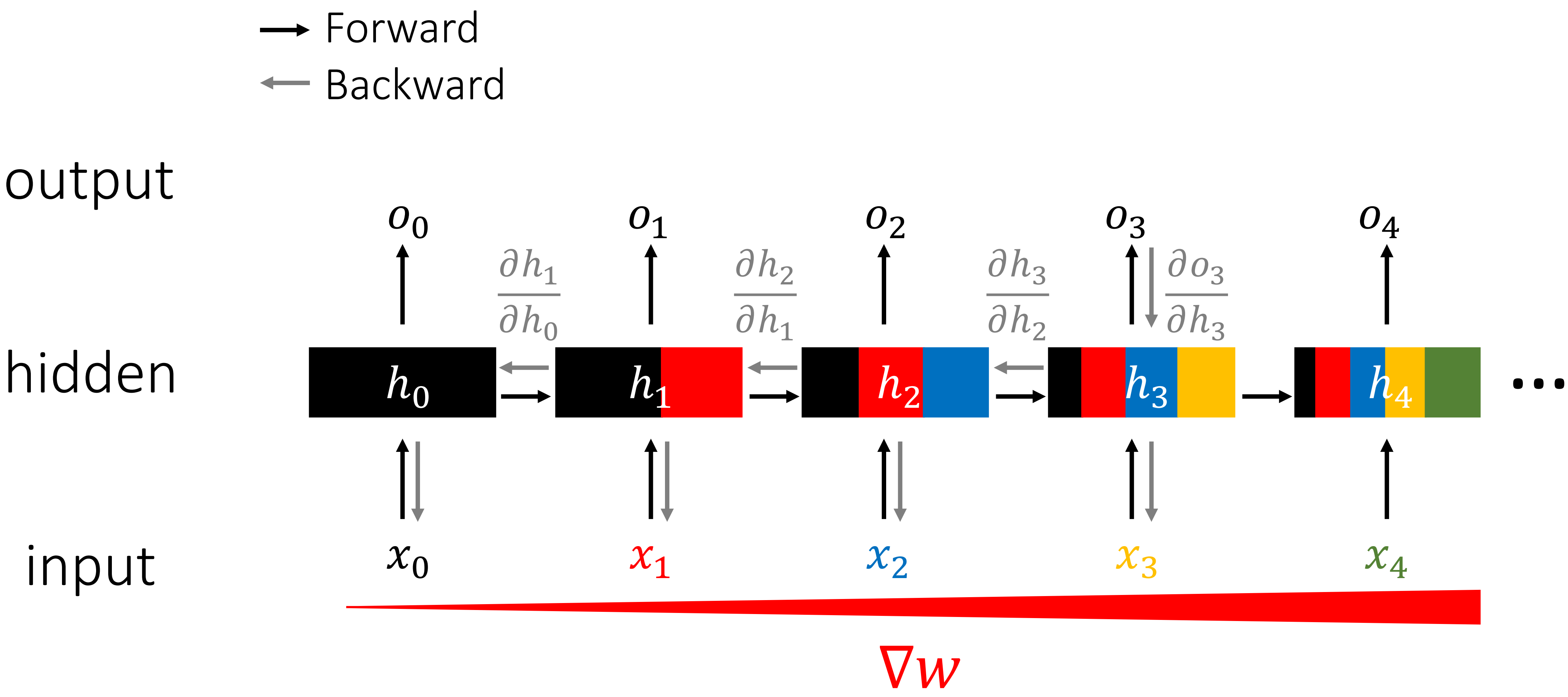


Recap: Backpropagation in RNNs

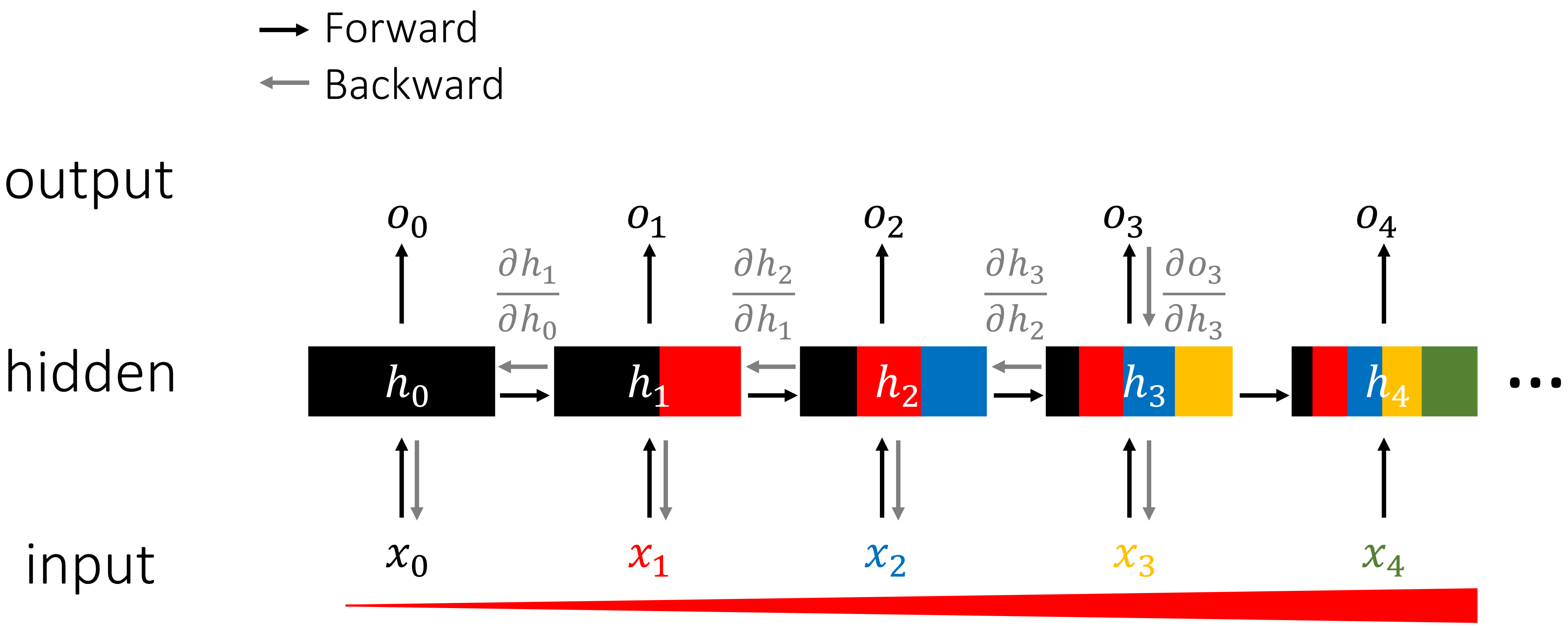


Backpropagation is performed backward in time

Vanishing and Exploding Gradients



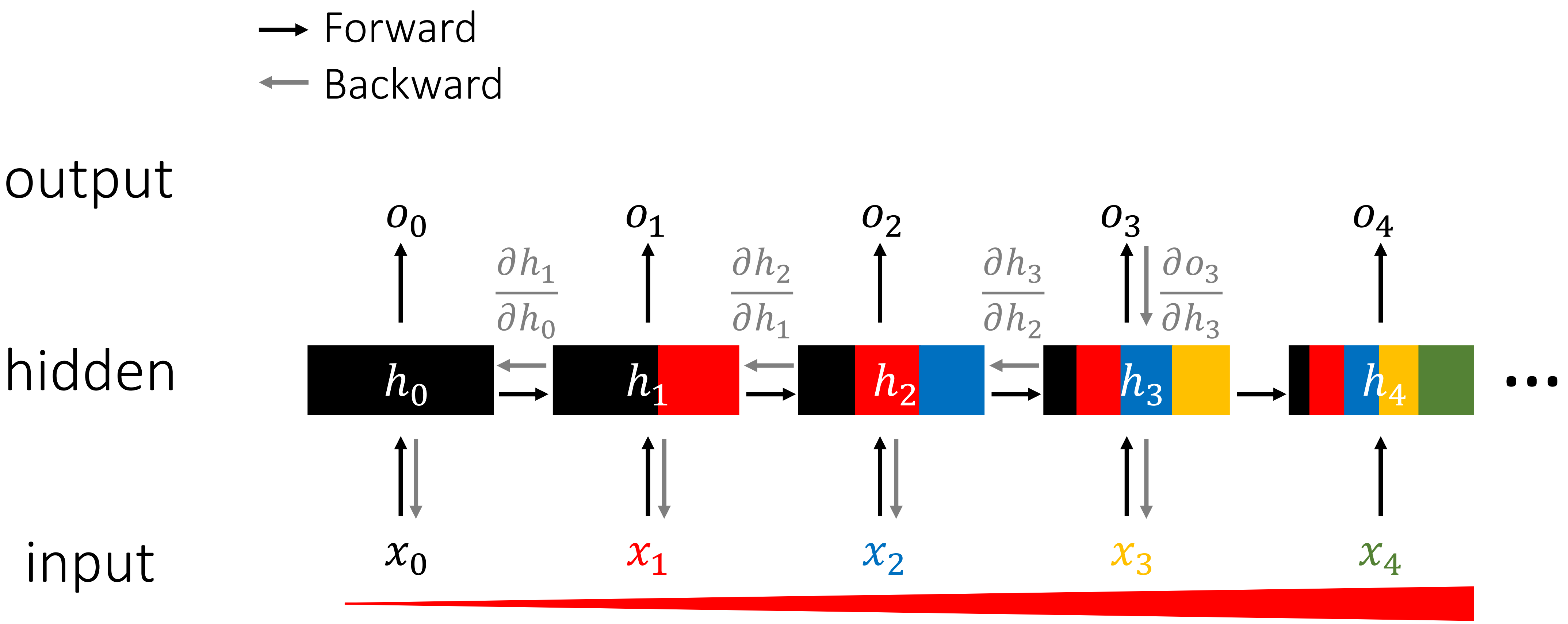
Vanishing and Exploding Gradients



Longer input sequence →

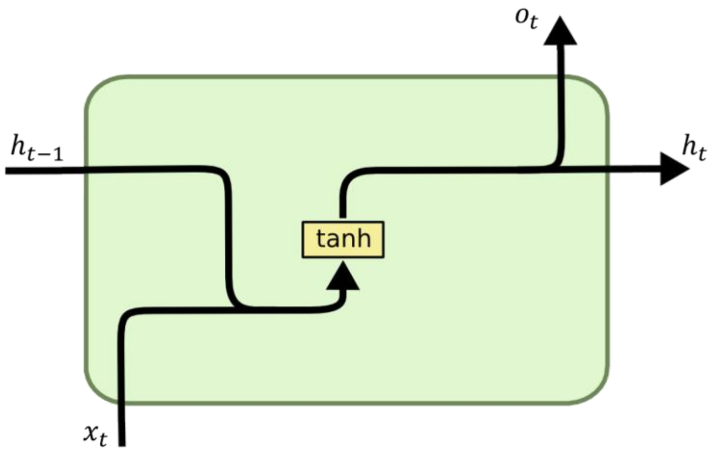
higher risk of Vanishing/Exploding Gradients!

Vanishing and Exploding Gradients

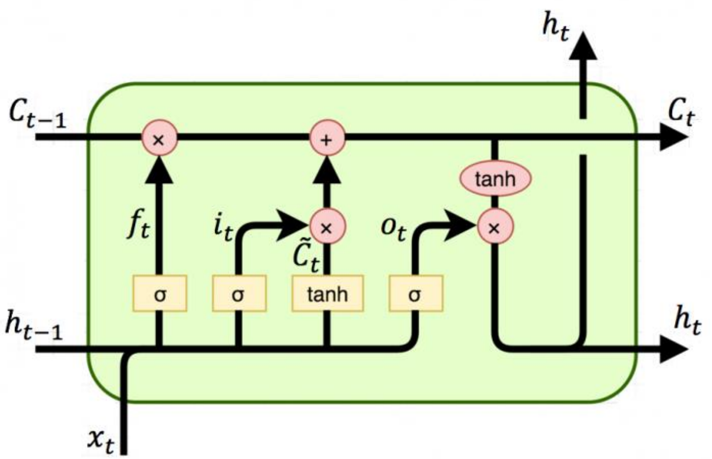


Need for better RNN architecture capable of processing longer sequence

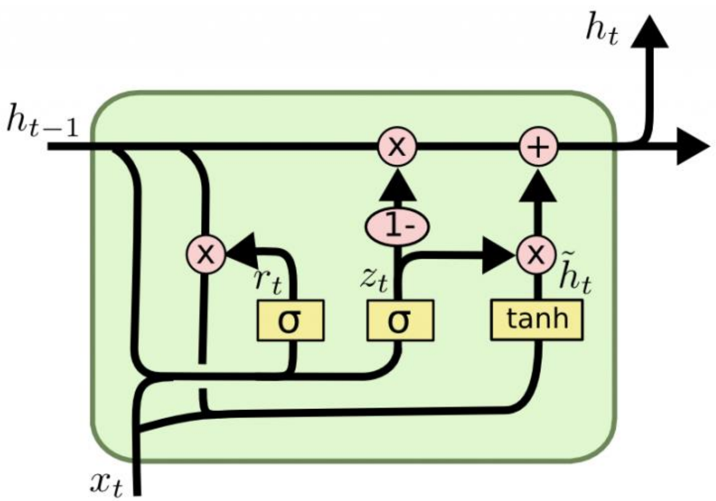
Gated RNNs



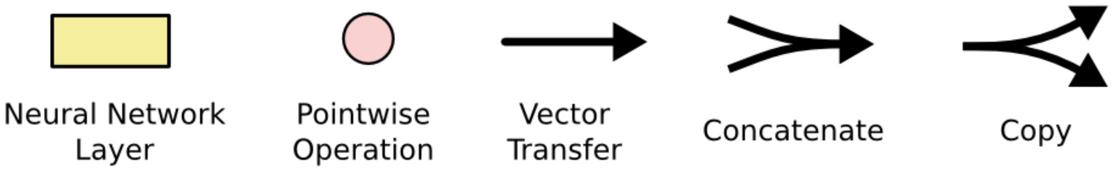
Vanilla RNN



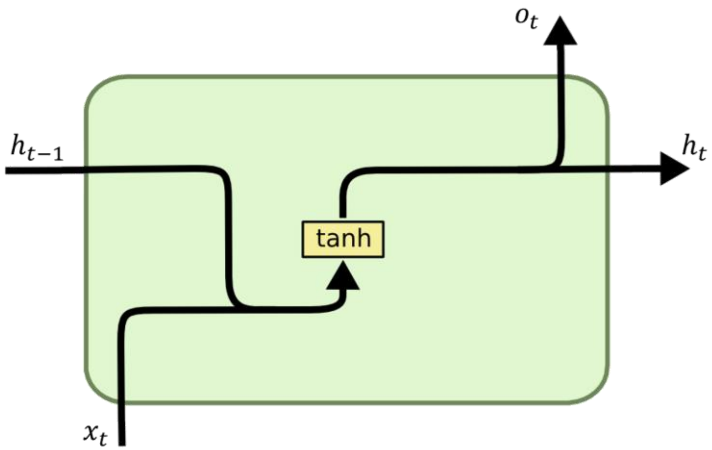
LSTM



GRU

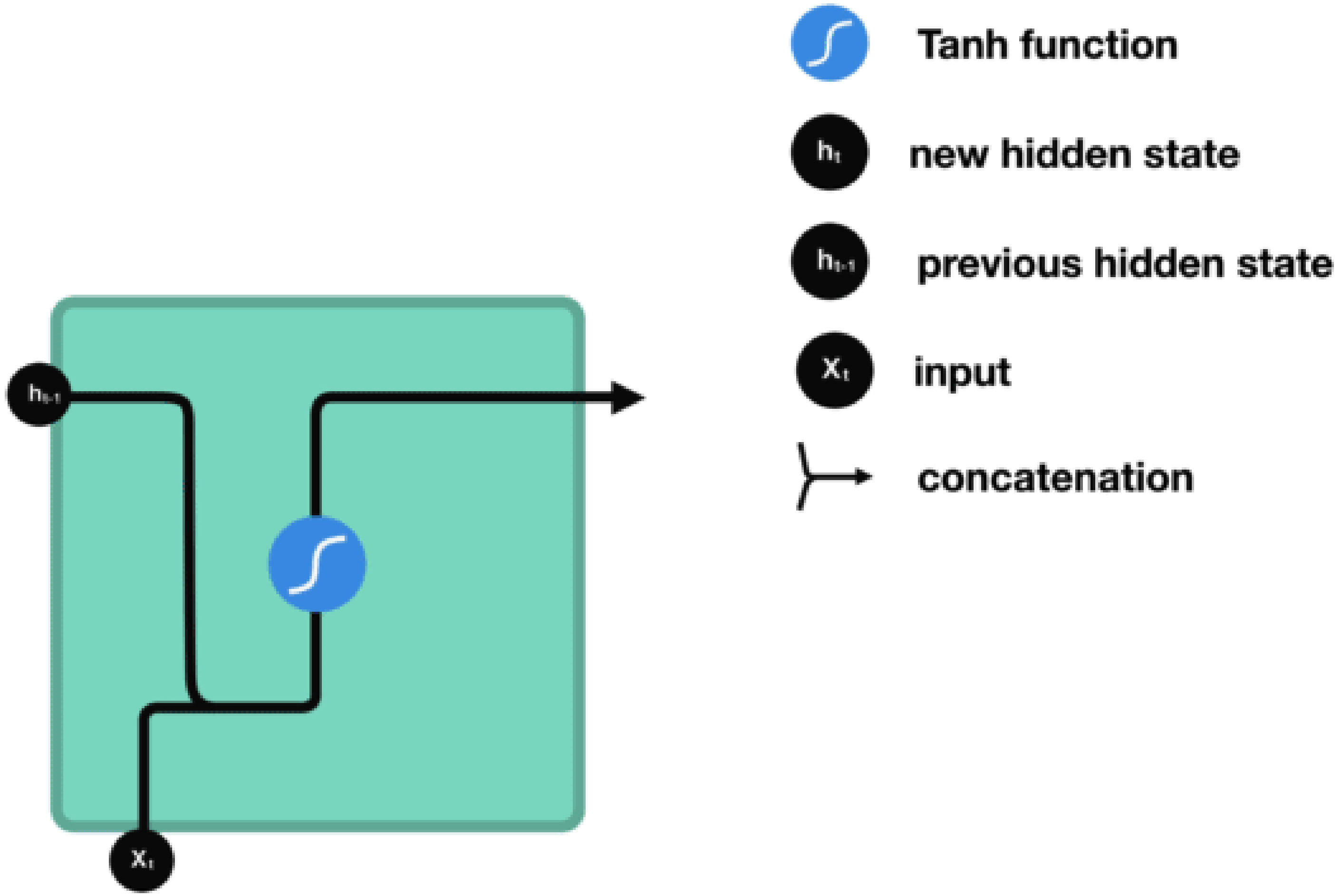


Vanilla RNN



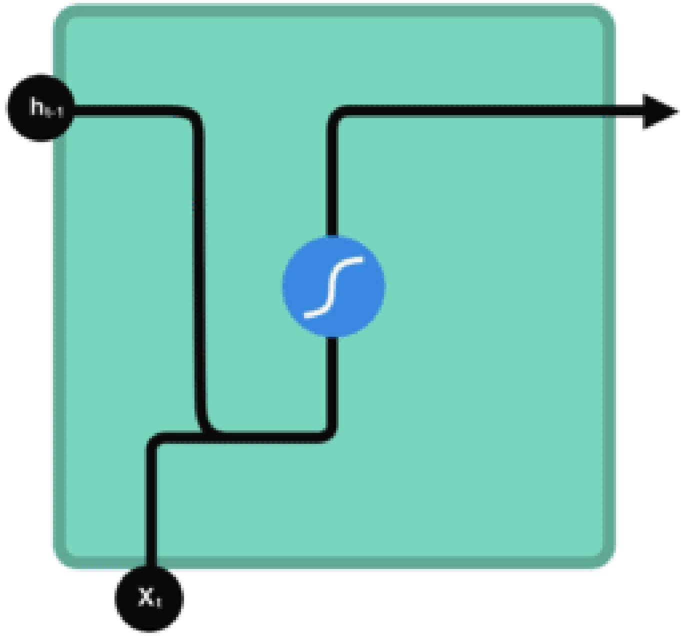
Vanilla RNN

Vanilla RNN



h_{t-1}
X_t
h_t
h_{t-1}
Tanh function
new hidden state
previous hidden state
input
concatenation

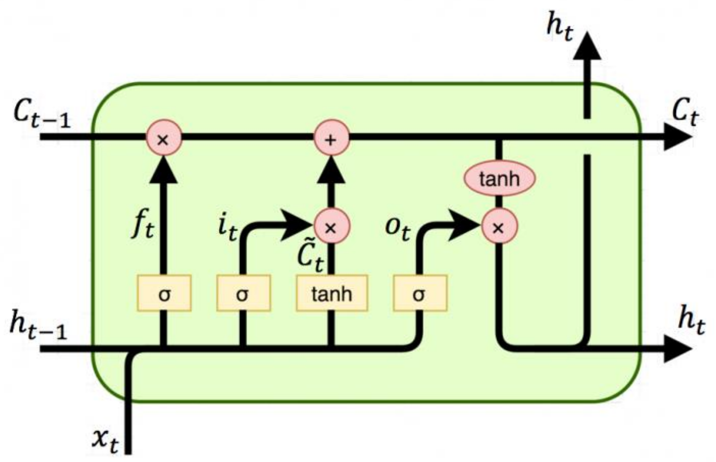
Vanilla RNN



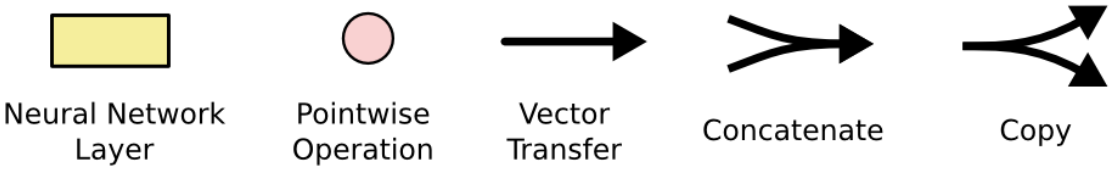
- Tanh function
- new hidden state
- previous hidden state
- input
- concatenation

$$
\begin{aligned}
\boldsymbol{a}^{(t)} &= \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)} \\
\boldsymbol{h}^{(t)} &= \tanh(\boldsymbol{a}^{(t)}) \\
\boldsymbol{o}^{(t)} &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)} \\
\hat{\boldsymbol{y}}^{(t)} &= \mathrm{softmax}(\boldsymbol{o}^{(t)})
\end{aligned}
$$

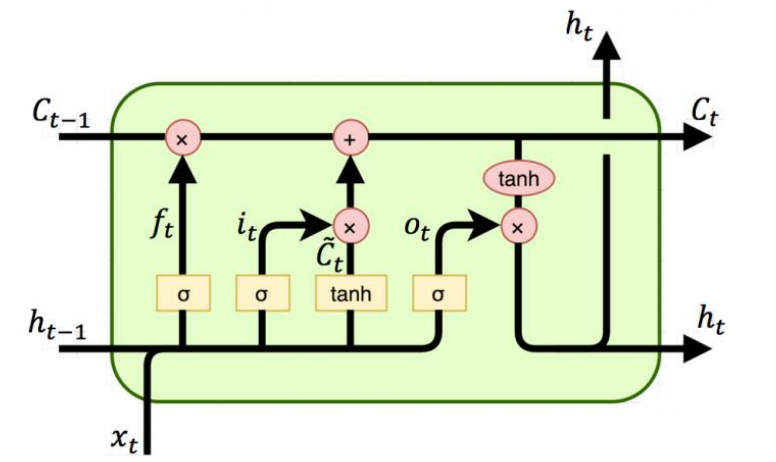
LSTM (Long Short-Term Memory)



LSTM

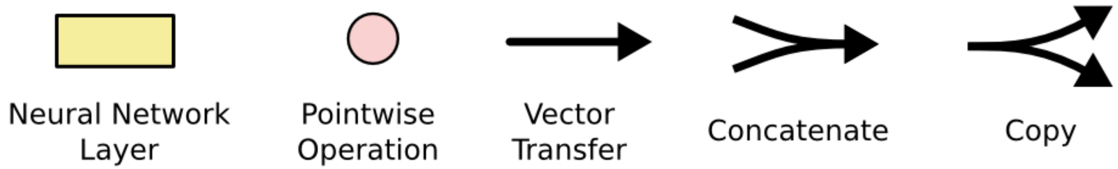


LSTM (Long Short-Term Memory)

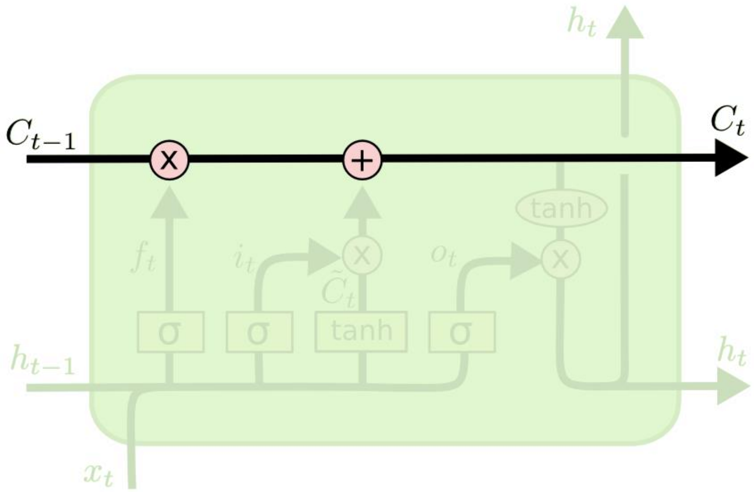


LSTM

$$
\begin{aligned}
f_t &= \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \\
i_t &= \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \\
o_t &= \sigma_g(W_o x_t + U_o h_{t-1} + b_o) \\
c_t &= f_t \circ c_{t-1} + i_t \circ \sigma_c(W_c x_t + U_c h_{t-1} + b_c) \\
h_t &= o_t \circ \sigma_h(c_t)
\end{aligned}
$$



LSTM: Detailed Architecture

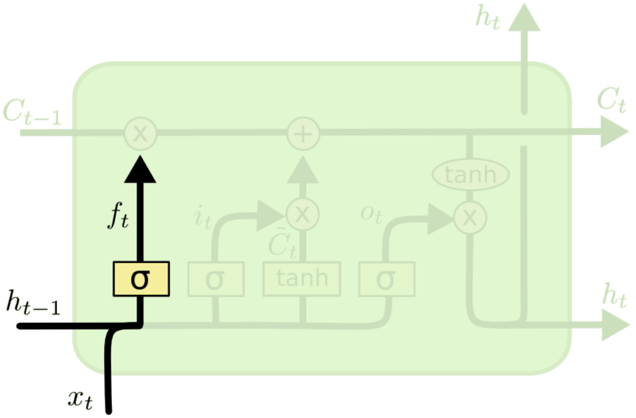


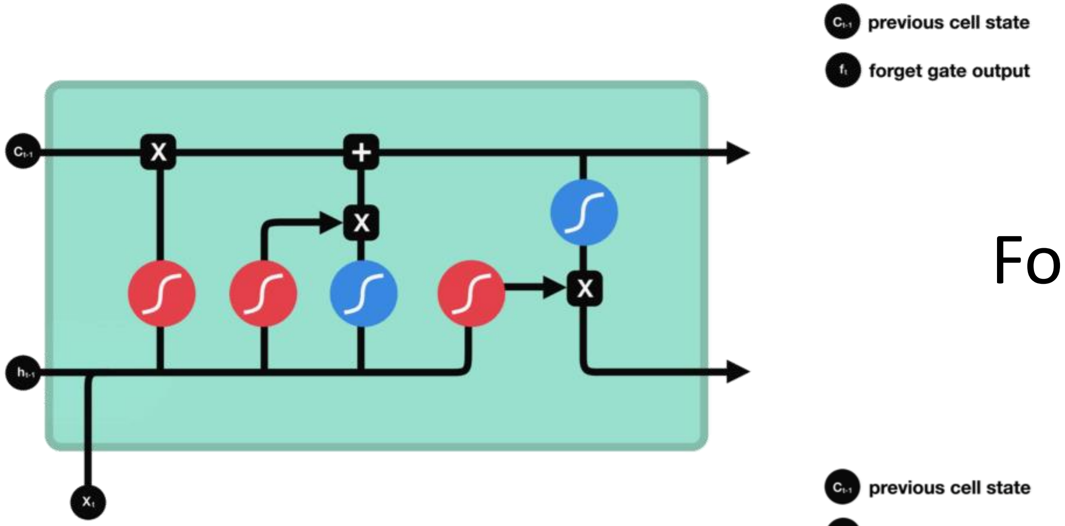
Cell state

- Unique to LSTM
- Long term memory of the model

LSTM: Detailed Architecture

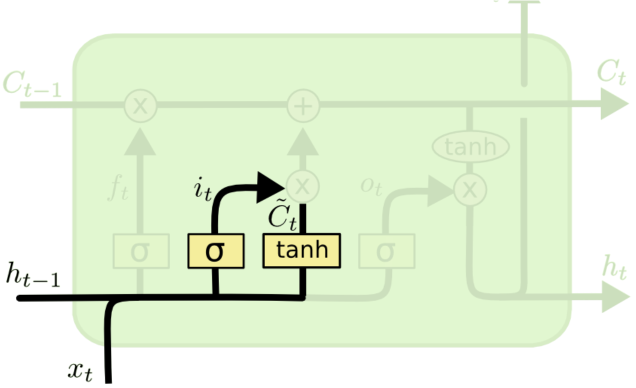
$$f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f)$$

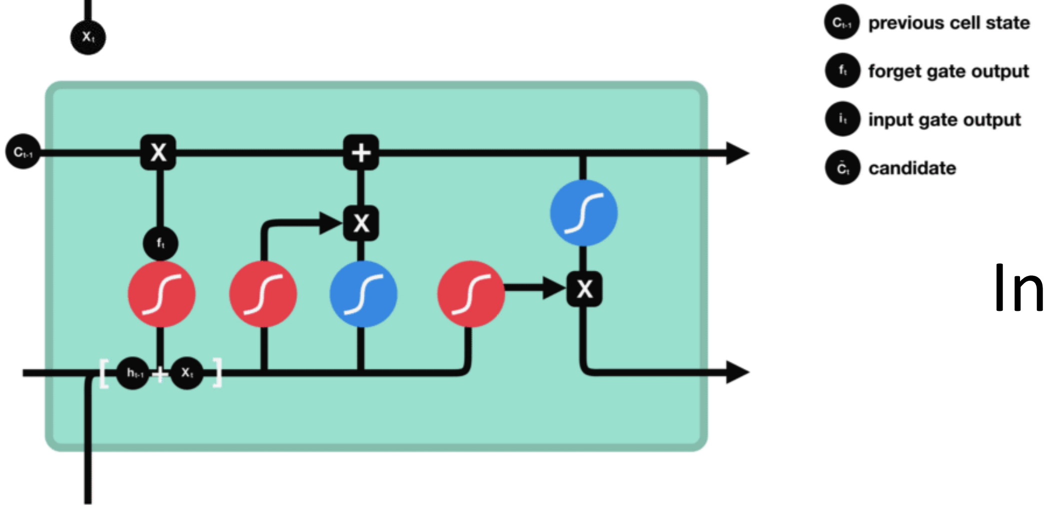




Forget gate layer

$$i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i)$$

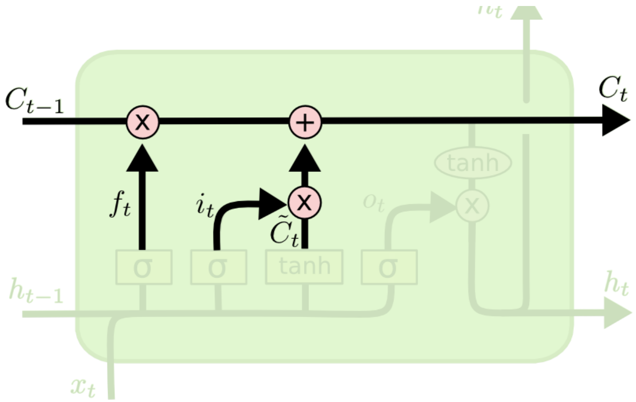


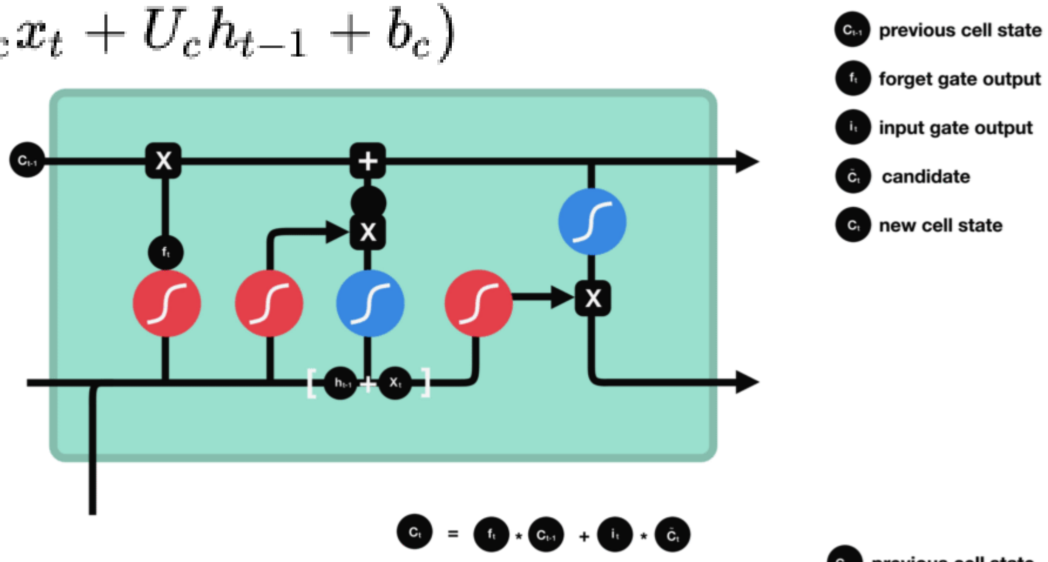


Input gate layer

LSTM: Detailed Architecture

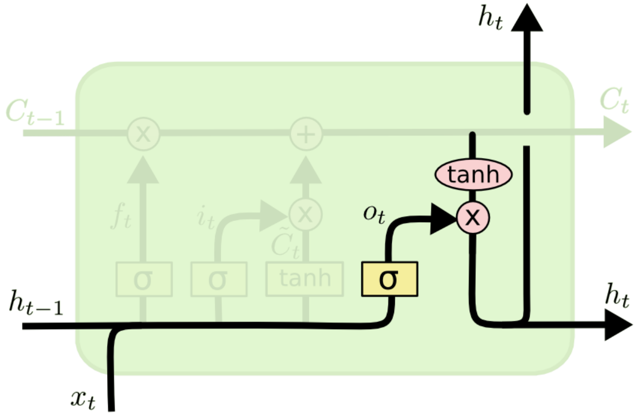
$$c_t = f_t \circ c_{t-1} + i_t \circ \sigma_c(W_c x_t + U_c h_{t-1} + b_c)$$

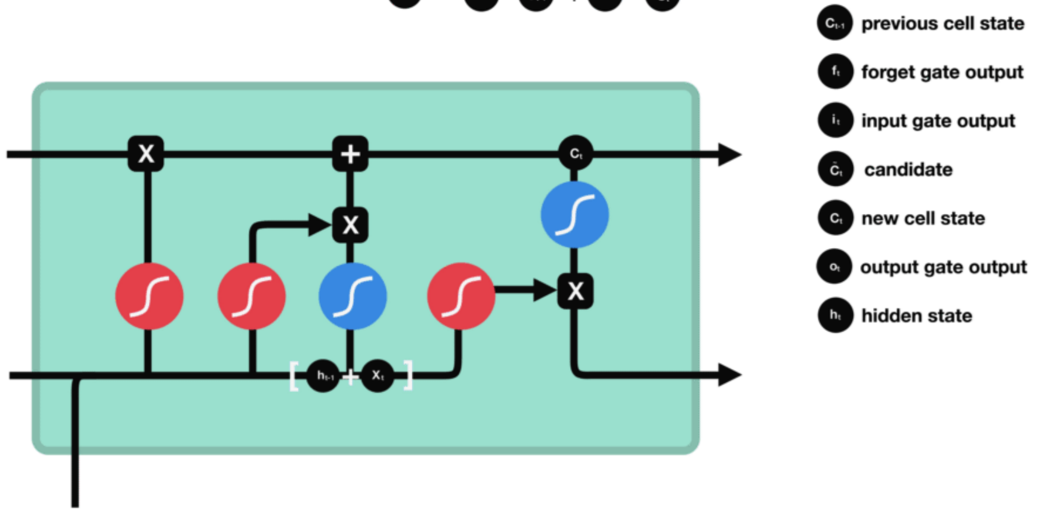




Update cell state

$$o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o)$$





Output gate layer

LSTM: Detailed Architecture

Forget gate

Decides what is relevant to keep from previous steps

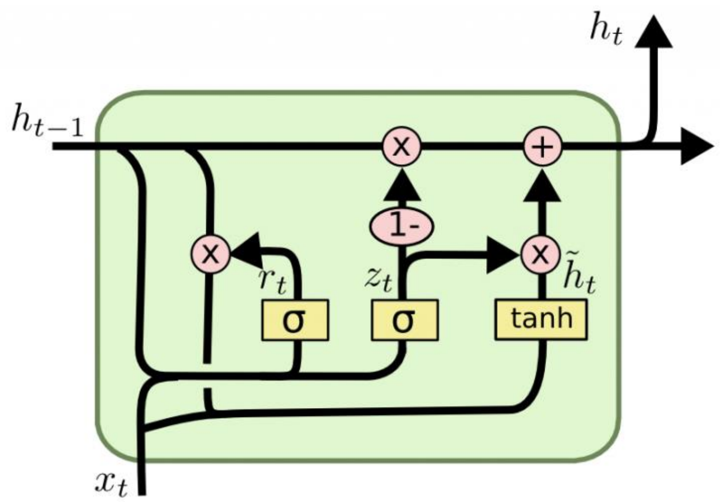
Input gate

Decides what information is relevant to add from the current step

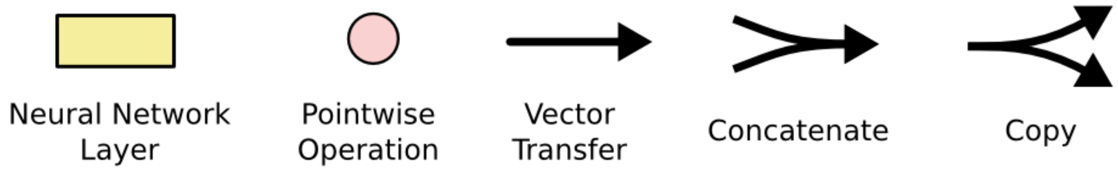
Output Gate

Determines what the next hidden state should be

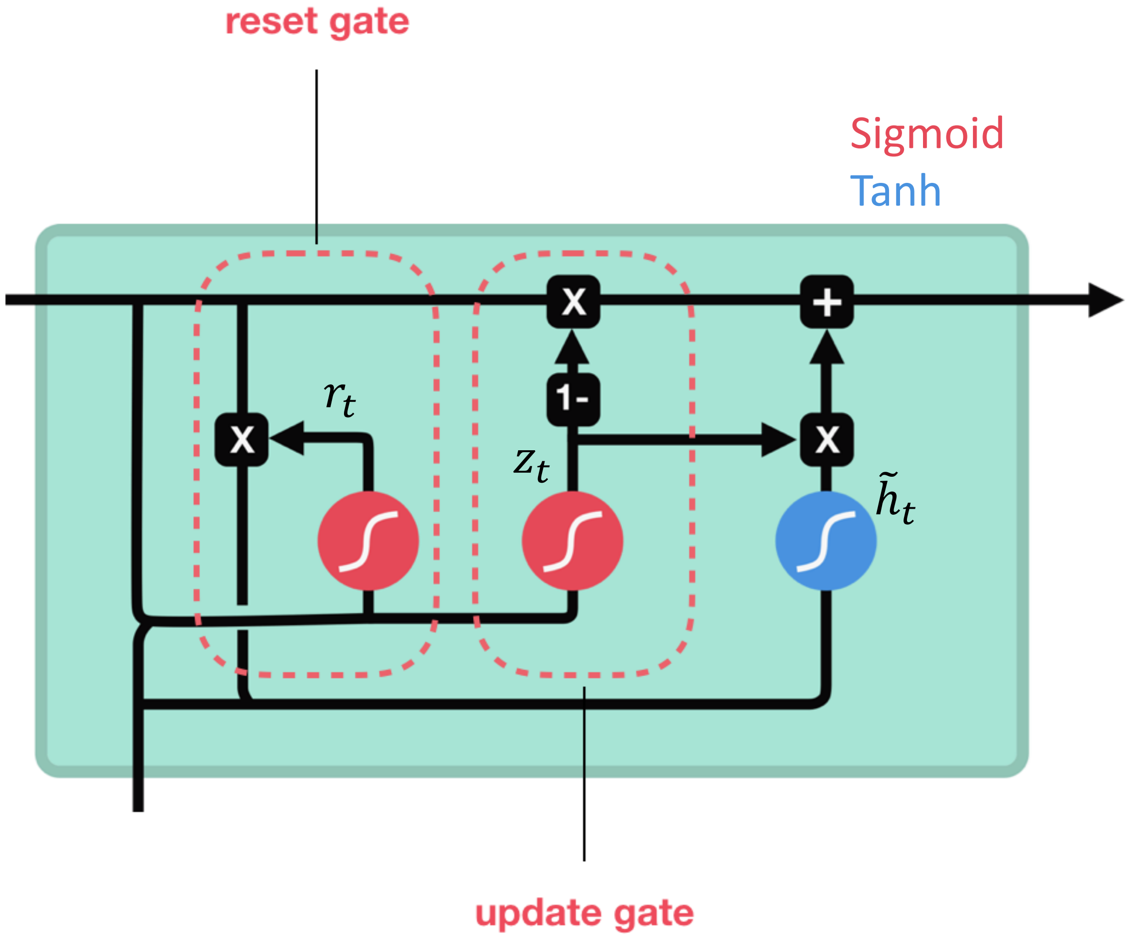
Gated RNNs



GRU



GRU: Detailed Architecture



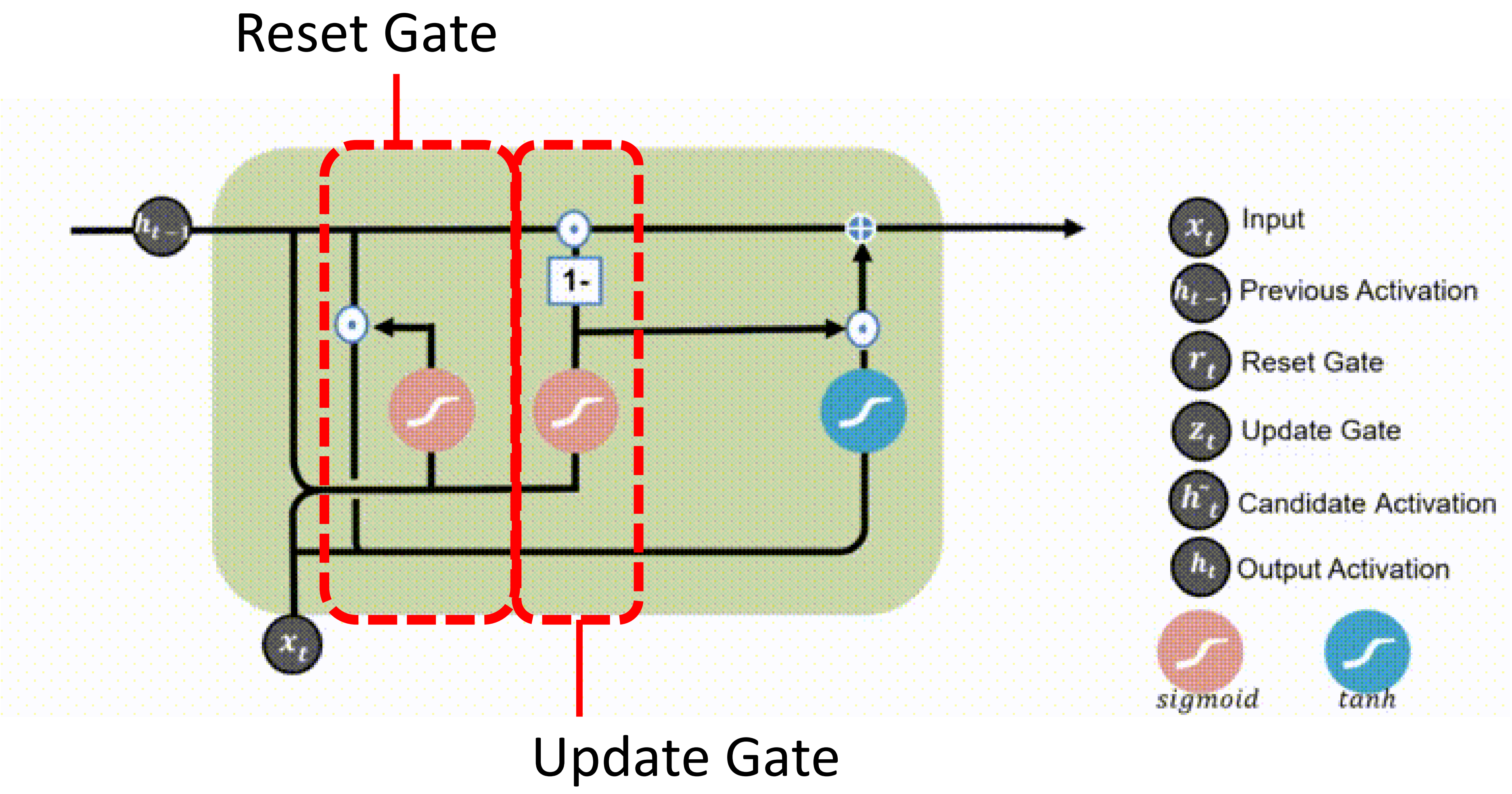
$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

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

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

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

Information Flow in GRU



GRU: Detailed Architecture

Update gate

How much of the past information needs to be retained

Reset gate

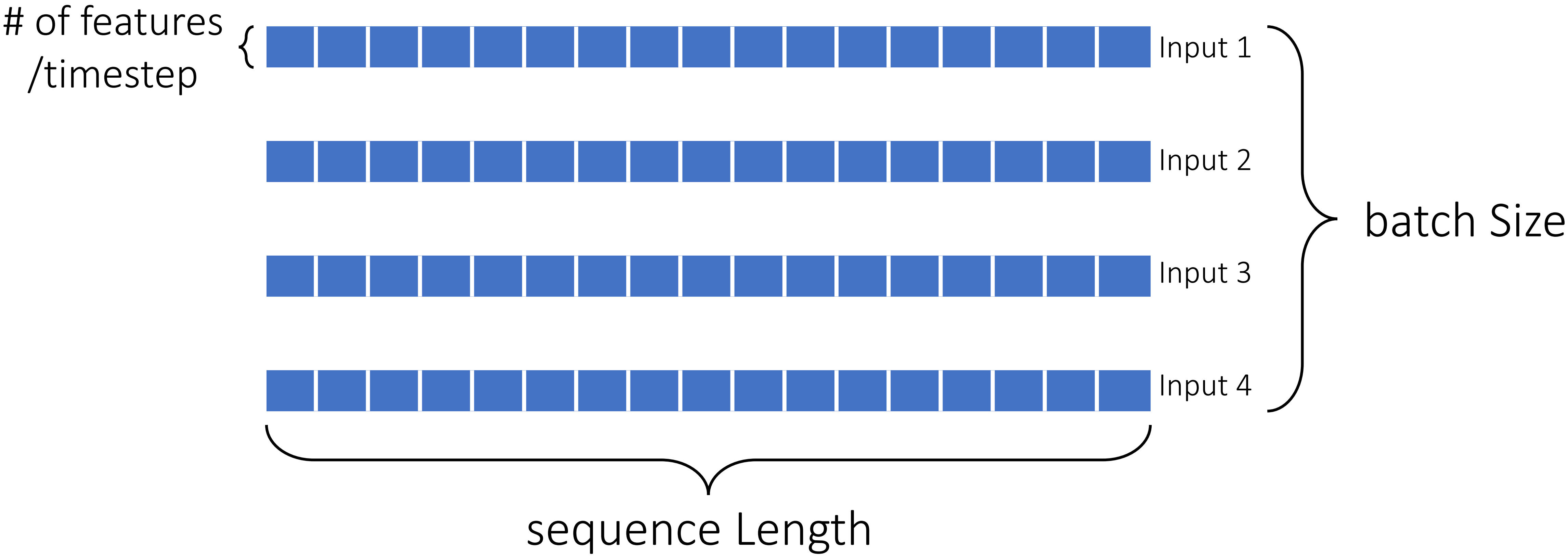
How much of the past information to forget

TRAINING GATED RNNs

Mini-batch Gradient in RNNs

RNN Extensions in LSTM/GRU

Mini-batch Gradient in RNNs

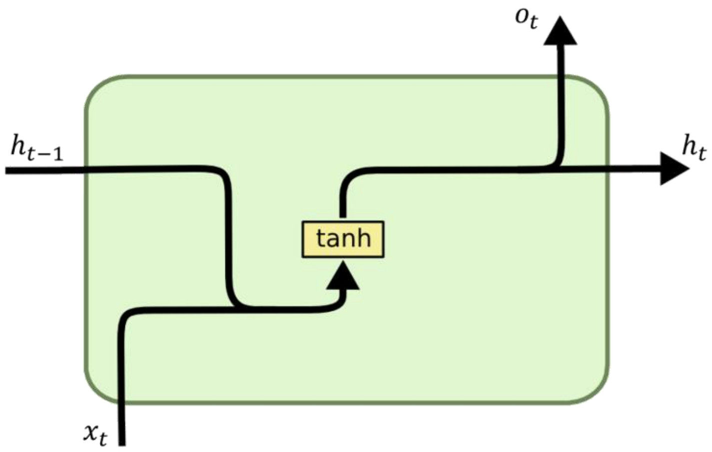


RNN input format in PyTorch = (batch size, sequence length, # of features)

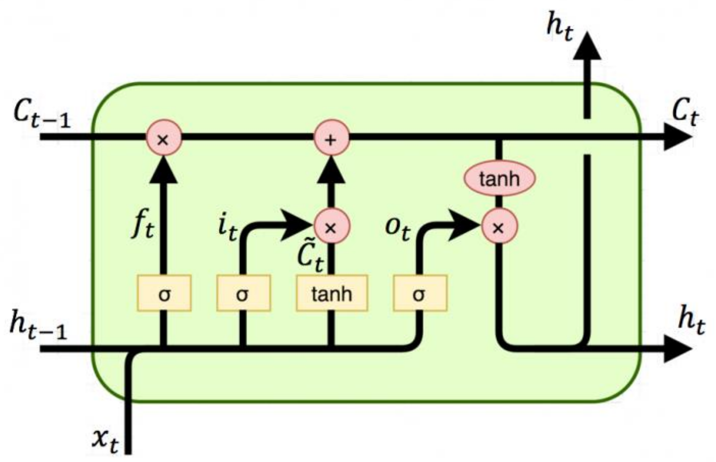
Example above = (4, 17, 1)

with batch_first = True

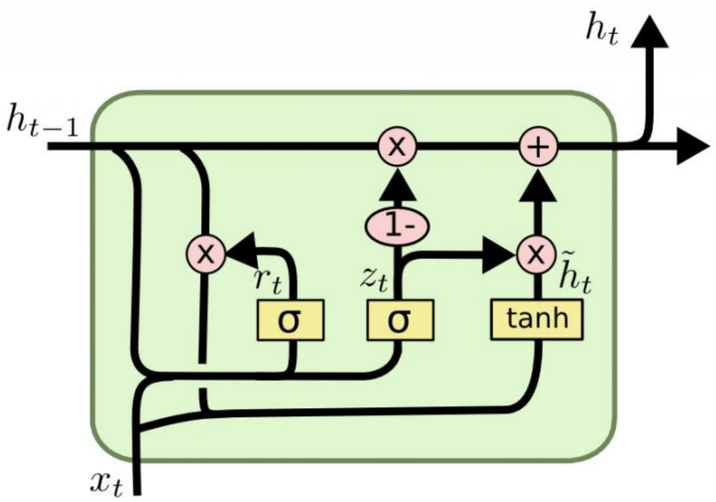
Gated RNNs



Vanilla RNN



LSTM

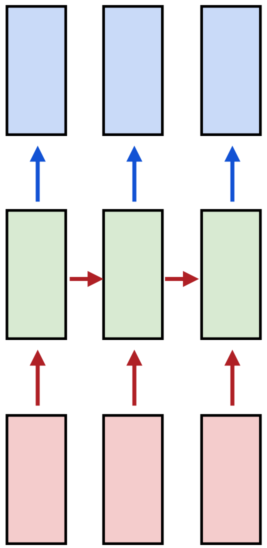


GRU

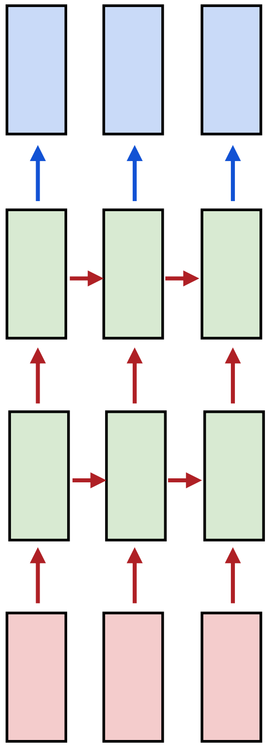
Inputs = x_t

Outputs = $f(h(t))$

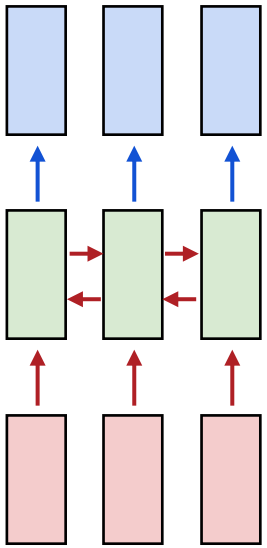
RNN Extensions in LSTM/GRU



Regular RNN



Deep RNN



Bi-directional RNN

RNN Extensions in LSTM/GRU

```python
class example_LSTM(torch.nn.Module):

    def __init__(self, input_size, hidden_size, num_layers, output_size):

        super(example_LSTM, self).__init__()

        self.lstm = torch.nn.LSTM(input_size=input_size, hidden_size=hidden_size,
                                  num_layers = num_layers,
                                  batch_first = True,
                                  bidirectional = False,
                                  dropout = 0.1)

        self.decoder = torch.nn.Linear(hidden_size, output_size)

    def forward(self, input_seq, hidden_state):

        pred, hidden = self.lstm(input_seq, hidden_state)

        pred = self.decoder(pred)

        return pred
```

Set to hidden_size * 2 if bidirectional = True

num_layers:
LSTM layers to be stacked

batch_first:
Tells PyTorch we are using (batchsize, seq_len, feature #)

bidirectional:
Whether to configure bidirectional LSTM

dropout:
introduces dropout layer on the outputs of each LSTM layer except for last layer (use when num_layers > 1)

RNN Extensions in LSTM/GRU

```python
class example_GRU(torch.nn.Module):

    def __init__(self, input_size, hidden_size, num_layers, output_size):

        super(example_GRU, self).__init__()

        self.gru = torch.nn.GRU(input_size=input_size, hidden_size=hidden_size,
                                num_layers = num_layers,
                                batch_first = True,
                                bidirectional = False,
                                dropout = 0.1)

        self.decoder = torch.nn.Linear(hidden_size, output_size)

    def forward(self, input_seq, hidden_state):

        pred, hidden = self.gru(input_seq, hidden_state)

        pred = self.decoder(pred)

        return pred
```

Set to hidden_size * 2 if bidirectional = True

num_layers:
GRU layers to be stacked

batch_first:
Tells PyTorch we are using (batchsize, seq_len, feature #)

bidirectional:
Whether to configure bidirectional GRU

dropout:
introduces dropout layer on the outputs of each GRU layer except for last layer
(use when num_layers > 1)

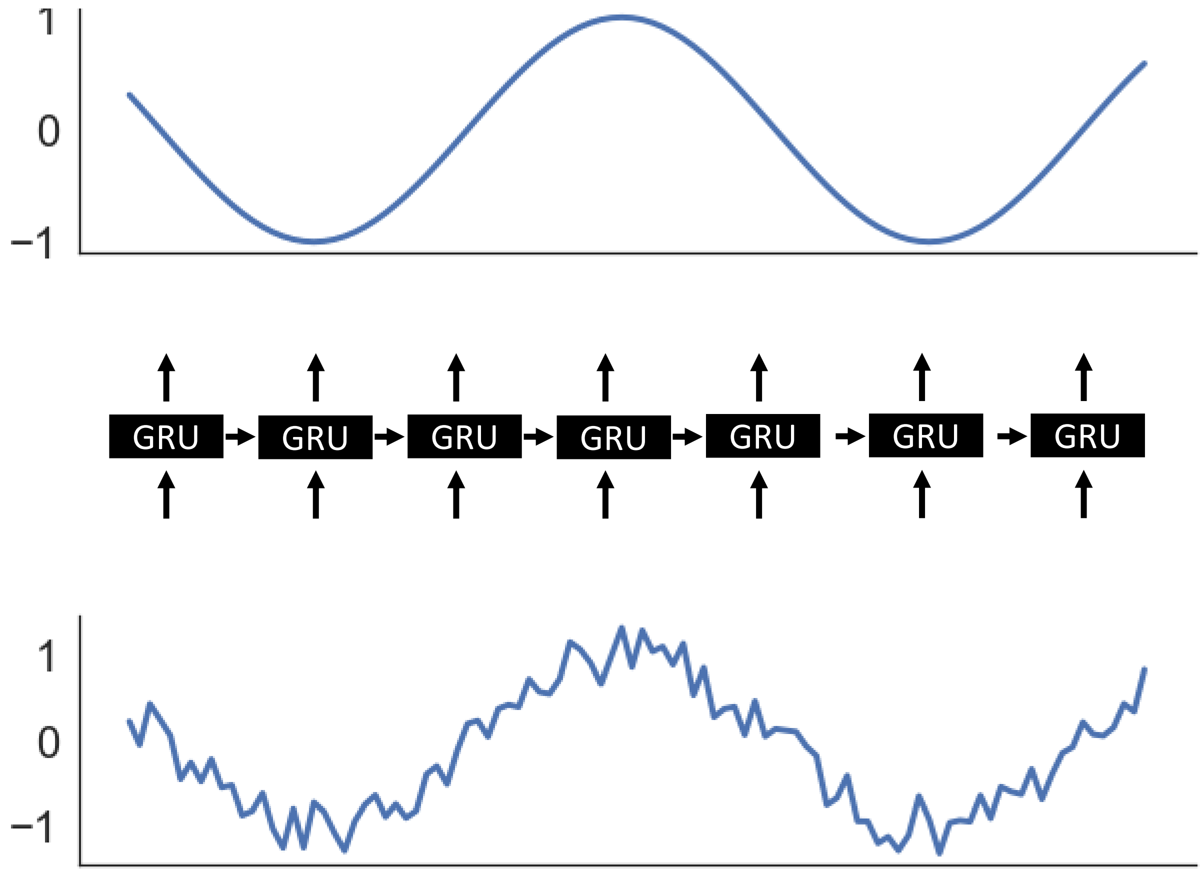
IMPLEMENTATION OF GATED RNNs in PYTORCH

Signal Denoising

Signal Denoising

Output Sequence

Input Sequence



Prepare Data

```python
def sinusoidal_generator(X, signal_freq=60.):

    return np.sin(2 * np.pi * (X) / signal_freq)

def add_noise(Y, noise_range=(-0.35, 0.35)):

    noise = np.random.uniform(noise_range[0], noise_range[1], size=Y.shape)

    return Y + noise

def sample_seq(sequence_length):

    random_offset = random.randint(0, sequence_length)
    X = np.arange(sequence_length)

    denoised_output_seq = sinusoidal_generator(X + random_offset)
    noisy_input_seq = add_noise(denoised_output_seq)

    return noisy_input_seq, denoised_output_seq
```

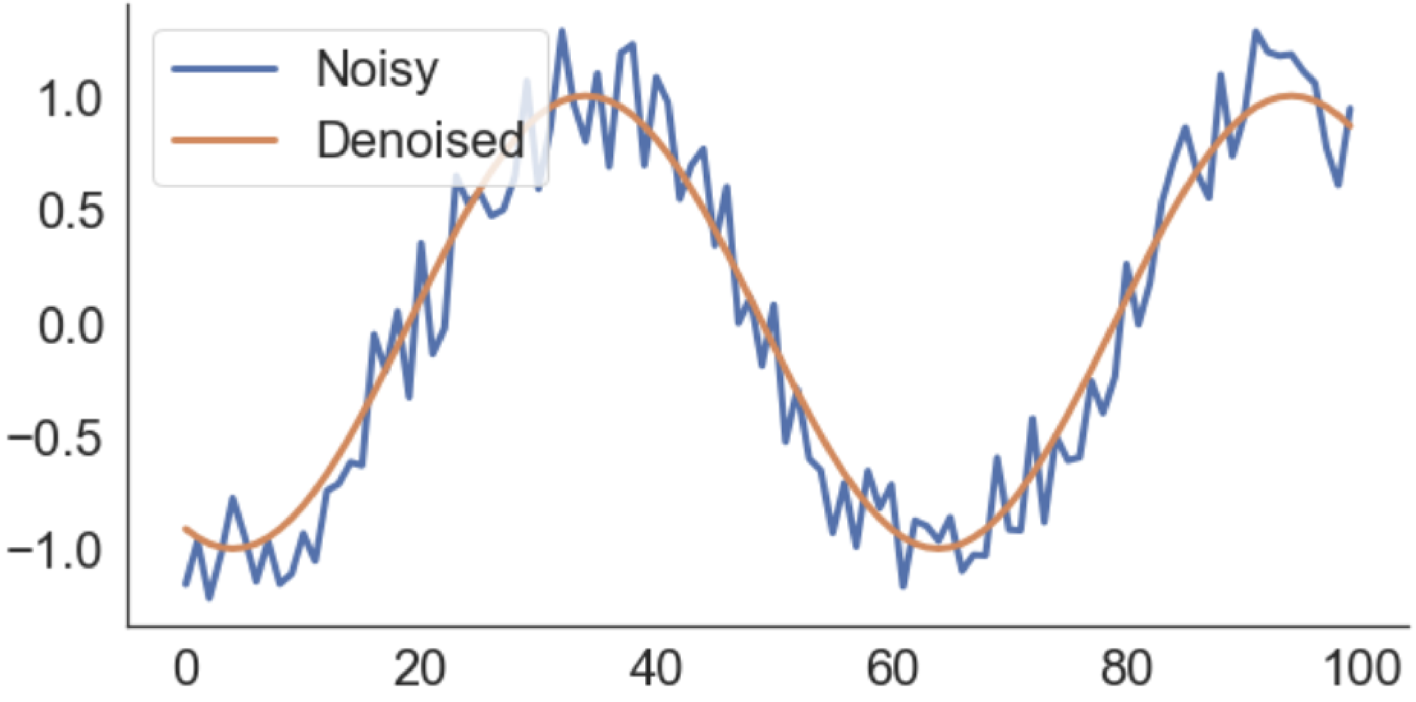
Sinusoidal wave generator

Add noise function

Generate sample ground truth/noisy sinusoidal waves

Prepare Data

```
noisy_input_seq, denoised_output_seq = sample_seq(sequence_length = 100)

plt.figure(figsize = (10, 5))

plt.plot(noisy_input_seq, label ='Noisy', linewidth = 3)
plt.plot(denoised_output_seq, label ='Denoised', linewidth = 3)
plt.legend()
sns.despine()
```



Example sample ground truth & noisy sinusoidal wave with sequence length = 100

Prepare Data

```python
def create_synthetic_dataset(n_samples, sequence_length):

    noisy_seq_inputs = np.zeros((n_samples, sequence_length))
    denoised_seq_outputs = np.zeros((n_samples, sequence_length))

    for i in range(n_samples):

        noisy_inp, denoised_out = sample_seq(sequence_length)

        noisy_seq_inputs[i, :] = noisy_inp
        denoised_seq_outputs[i, :] = denoised_out

    return noisy_seq_inputs, denoised_seq_outputs
```

Using the sample_seq() function to generate synthetic ground truth/noisy dataset of n-samples

```python
noisy_seq_inputs, denoised_seq_outputs = create_synthetic_dataset(n_samples = 12000,
                                                                  sequence_length = 100)

train_input_seqs, train_output_seqs = noisy_seq_inputs[:8000], denoised_seq_outputs[:8000]
test_input_seqs, test_output_seqs = noisy_seq_inputs[8000:], denoised_seq_outputs[8000:]
```

Take first 8000 as training dataset and 4000 as testing dataset

```python
train_input_seqs = train_input_seqs.reshape((train_input_seqs.shape[0], -1, 1))
train_output_seqs = train_output_seqs.reshape((train_output_seqs.shape[0], -1, 1))

test_input_seqs = test_input_seqs.reshape((test_input_seqs.shape[0], -1, 1))
test_output_seqs = test_output_seqs.reshape((test_output_seqs.shape[0], -1, 1))
```

Reshape training and testing dataset to conform to (# of samples, seq_len, feature #) format

Define Model

```python
class Denoiser_GRU(torch.nn.Module):

    def __init__(self, input_size, hidden_size, num_layers, output_size):

        super(Denoiser_GRU, self).__init__()

        self.gru = torch.nn.GRU(input_size=input_size, hidden_size=hidden_size,
                                num_layers = num_layers,
                                batch_first = True,
                                bidirectional = False)

        self.decoder = torch.nn.Linear(hidden_size, output_size)

        self.output_activation = torch.nn.Tanh()

    def forward(self, input_seq, hidden_state):

        pred, hidden = self.gru(input_seq, hidden_state)

        pred = self.output_activation(self.decoder(pred))

        return pred
```

Using GRU with batch_first = True

Decoder layer to convert hidden states to final output

Using **Tanh** on decoder output layer to squeeze output value between -1 and 1

Input_sequence, hidden_states → GRU → output_sequence, hidden_states → Decoder Layer → Tanh activation

Define Hyperparameters

```python
denoiser_GRU = Denoiser_GRU(input_size = 1, hidden_size = 30,
                            num_layers = 1, output_size = 1)

learning_rate = 0.0003
epochs = 100

batchsize = 300

loss_func = torch.nn.L1Loss()
optimizer = torch.optim.Adam(denoiser_GRU.parameters(), lr=learning_rate)

denoiser_GRU
```

Input dim to GRU = 1
Hidden state size = 30
GRU layers to be stacked = 1
Output dim of decoder layer = 1

Define learning rate, epochs and batch size

Using L1Loss (Least Absolute Deviations) and Adam optimizer

Identify Tracked Values

```
train_loss_list = []
```

Empty Python list to keep track of training loss

Train Model

```python
train_input_seqs = torch.from_numpy(train_input_seqs).float()
train_output_seqs = torch.from_numpy(train_output_seqs).float()

test_input_seqs = torch.from_numpy(test_input_seqs).float()
test_output_seqs = torch.from_numpy(test_output_seqs).float()

train_batches_features = torch.split(train_input_seqs, batchsize)
train_batches_targets = torch.split(train_output_seqs, batchsize)

batch_split_num = len(train_batches_features)

for epoch in range(epochs):

    for k in range(batch_split_num):

        hidden_state = None

        pred = denoiser_GRU(train_batches_features[k], hidden_state)

        optimizer.zero_grad()

        loss = loss_func(pred, train_batches_targets[k])
        train_loss_list.append(loss.item())

        loss.backward()

        optimizer.step()

    print("Averaged Training Loss for Epoch ", epoch,": ", np.mean(train_loss_list[-batch_split_num:]))
```

Convert training and testing data to Tensors

Split training data into mini-batches

Training loop with mini-batch gradient

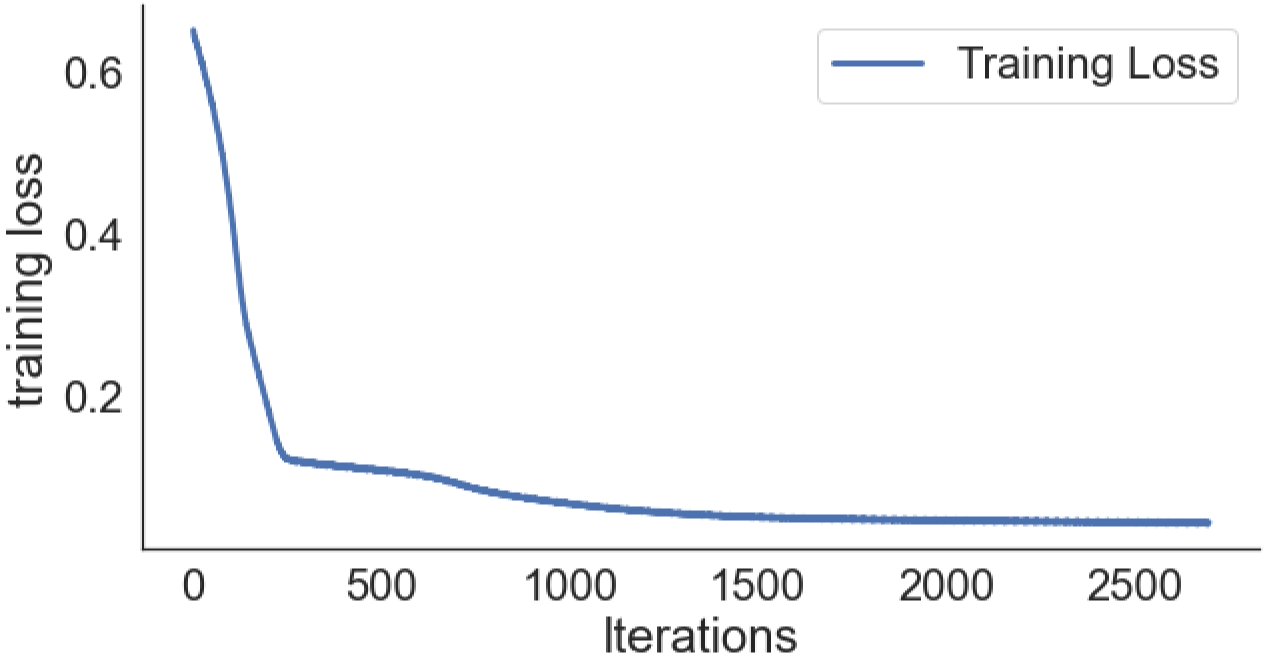
Print averaged loss throughout the epoch

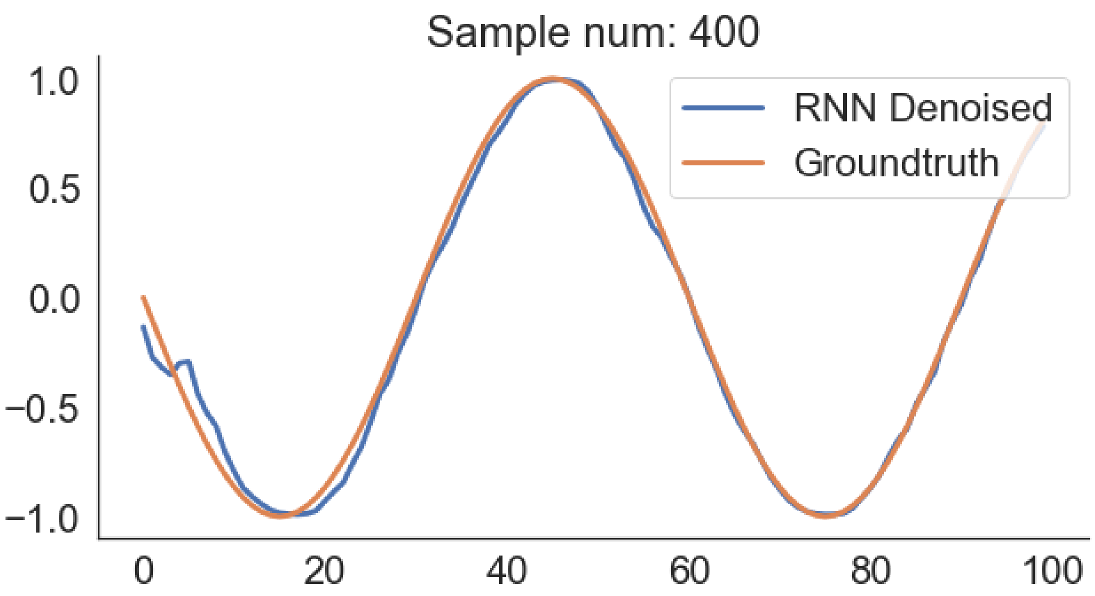
Visualize & Evaluate Model

```python
plt.figure(figsize = (10, 5))

plt.plot(train_loss_list, linewidth = 3, label = 'Training Loss')
plt.ylabel("training loss")
plt.xlabel("Iterations")
plt.legend()
sns.despine()
```

```python
with torch.no_grad():

    test_prediction = denoiser_GRU(test_input_seqs, None)
    print("Testing Loss (Least Absolute Deviations): ",
          loss_func(test_prediction, test_output_seqs).item())
```

```
Testing Loss (Least Absolute Deviations):  0.04158925637602806
```

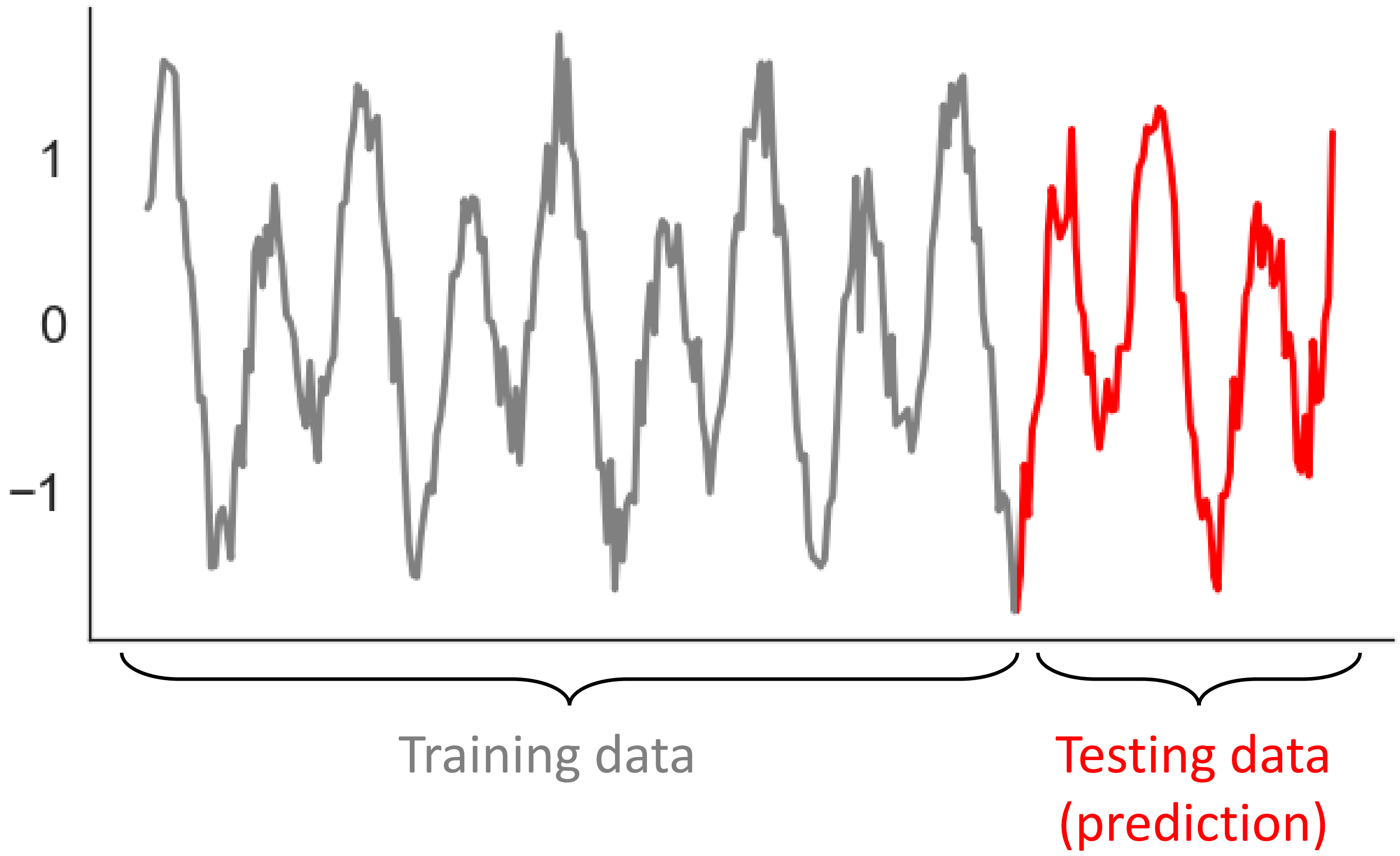




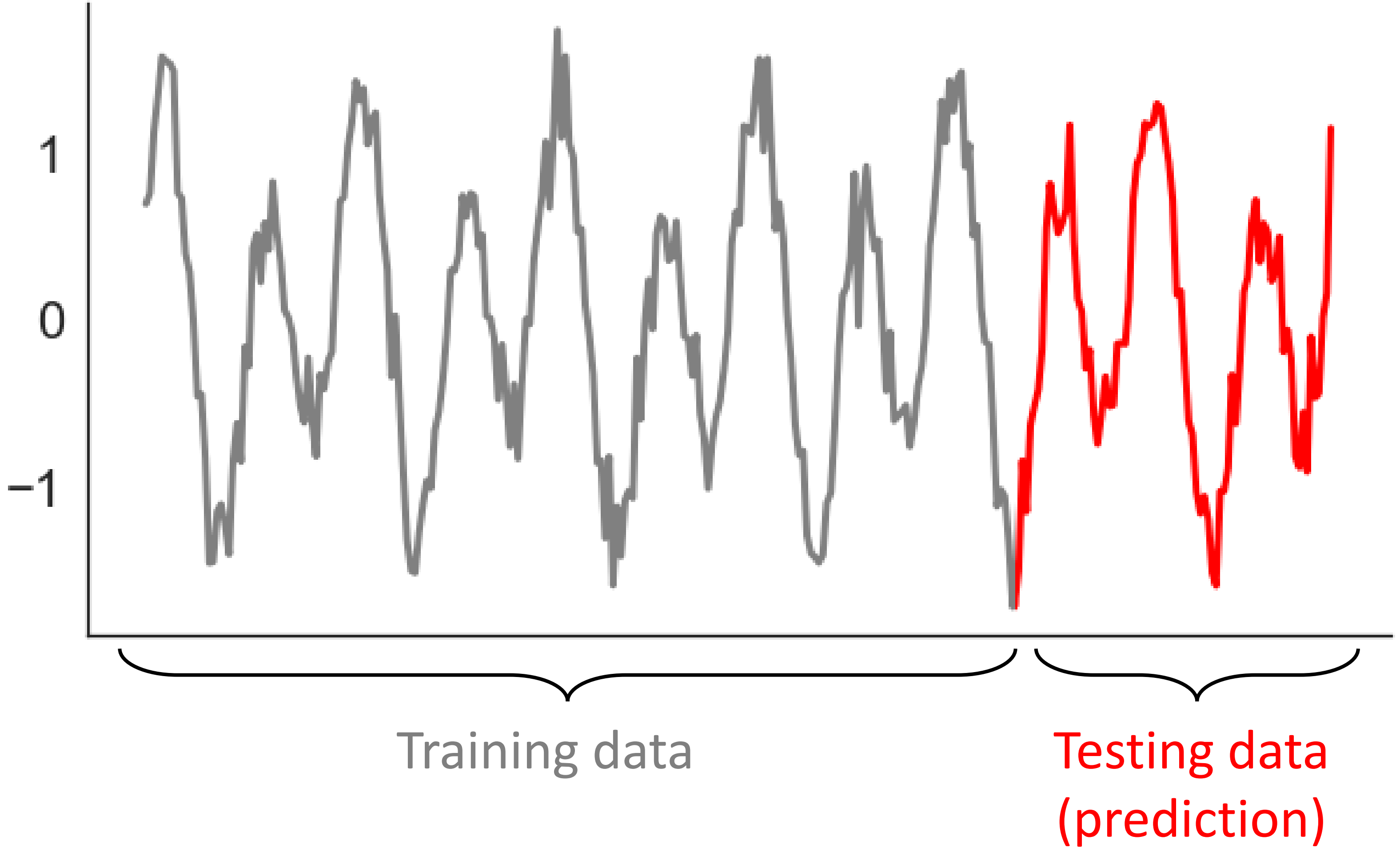
ENCODER-DECODER APPLICATION IN PYTORCH

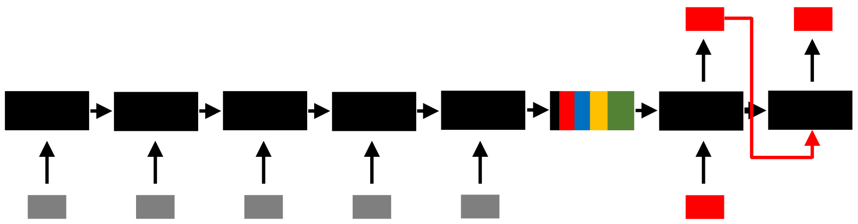
Signal Prediction

Example Task Description



Example Task Description





Encoder input sequence length = 5

Decoder output sequence length = 2
(prediction)

Prepare Data

```python
def generate_noisy_signal(datapoints_num, tf):

    t = np.linspace(0., tf, datapoints_num)
    y = np.sin(2. * t) + 0.5 * np.cos(t) + np.random.normal(0., 0.2, datapoints_num)

    return y.reshape(-1, 1)
```

Function for generating a noisy signal (sin + cos + noise)

```python
def generate_input_output_seqs(y, encoder_inputseq_len, decoder_outputseq_len, stride = 1, num_features = 1):

    L = y.shape[0]
    num_samples = (L - encoder_inputseq_len - decoder_outputseq_len) // stride + 1

    train_input_seqs = np.zeros([num_samples, encoder_inputseq_len, num_features])
    train_output_seqs = np.zeros([num_samples, decoder_outputseq_len, num_features])

    for ff in np.arange(num_features):

        for ii in np.arange(num_samples):

            start_x = stride * ii
            end_x = start_x + encoder_inputseq_len
            train_input_seqs[ii, :, ff] = y[start_x:end_x, ff]

            start_y = stride * ii + encoder_inputseq_len
            end_y = start_y + decoder_outputseq_len
            train_output_seqs[ii, :, ff] = y[start_y:end_y, ff]

    return train_input_seqs, train_output_seqs
```

Function for generating

- input sequences to encoder
- output target sequences for decoder

e.g., y = [1,2,3,4,5,6,7,8]
Encoder inputseq len = 3
Decoder outputseq len = 2

train_input_seqs =
[[1,2,3],[2,3,4],[3,4,5],[4,5,6]]
train_output_seqs =
[[4,5],[5,6],[6,7],[7,8]]

Prepare Data

```python
encoder_inputseq_len = 5
decoder_outputseq_len = 2
testing_sequence_len = 50

y = generate_noisy_signal(datapoints_num = 2000, tf = 80 * np.pi)
y_train = y[:-testing_sequence_len]
```

- Encoder input sequence length = 5
- Decoder output sequence length = 2
- Testing sequence length = 50

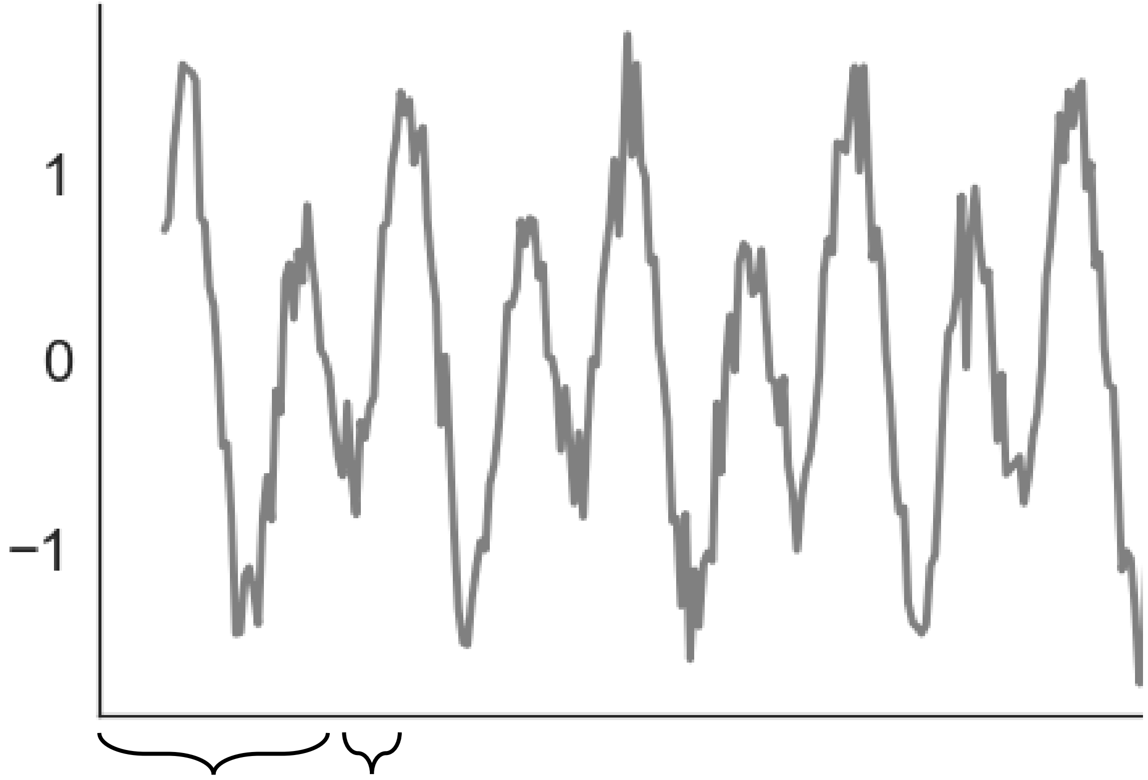
```python
train_input_seqs, train_output_seqs = generate_input_output_seqs(y = y_train,
                                                                 encoder_inputseq_len = encoder_inputseq_len,
                                                                 decoder_outputseq_len = decoder_outputseq_len,
                                                                 stride = 1,
                                                                 num_features = 1)
```

```python
print("Encoder Training Inputs Shape: ", train_input_seqs.shape)
print("Decoder Training Outputs Shape: ", train_output_seqs.shape)
```

```
Encoder Training Inputs Shape:  (1944, 5, 1)
Decoder Training Outputs Shape:  (1944, 2, 1)
```

(sample size, sequence length, feature/timestep)

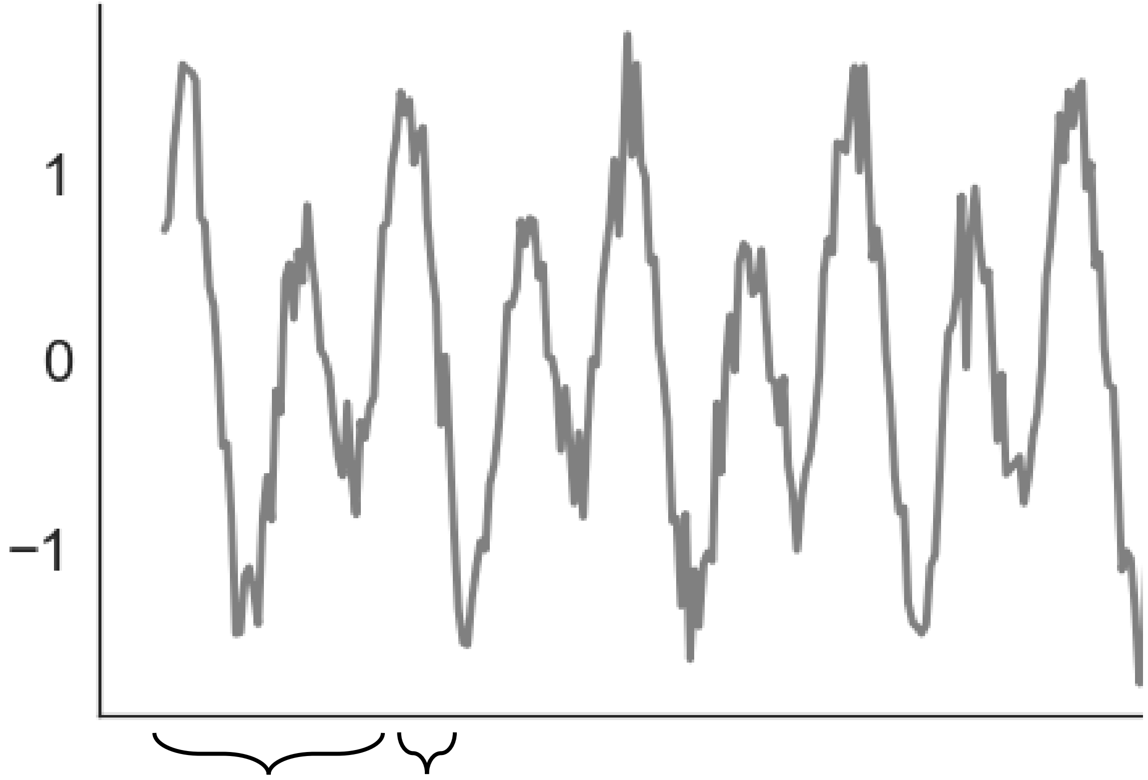
Prepare Data



train_input_seqs[0]
(input to encoder)

train_output_seqs[0]
(output target by decoder)

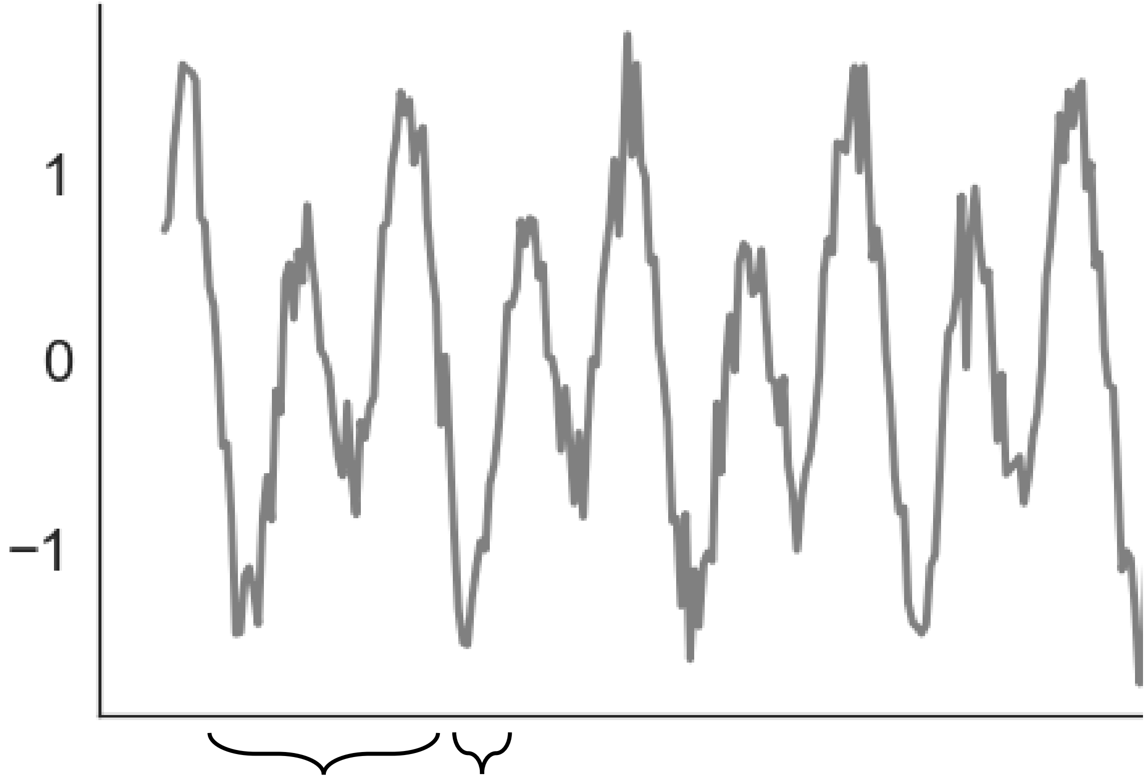
Prepare Data



train_input_seqs[1]

train_output_seqs[1]

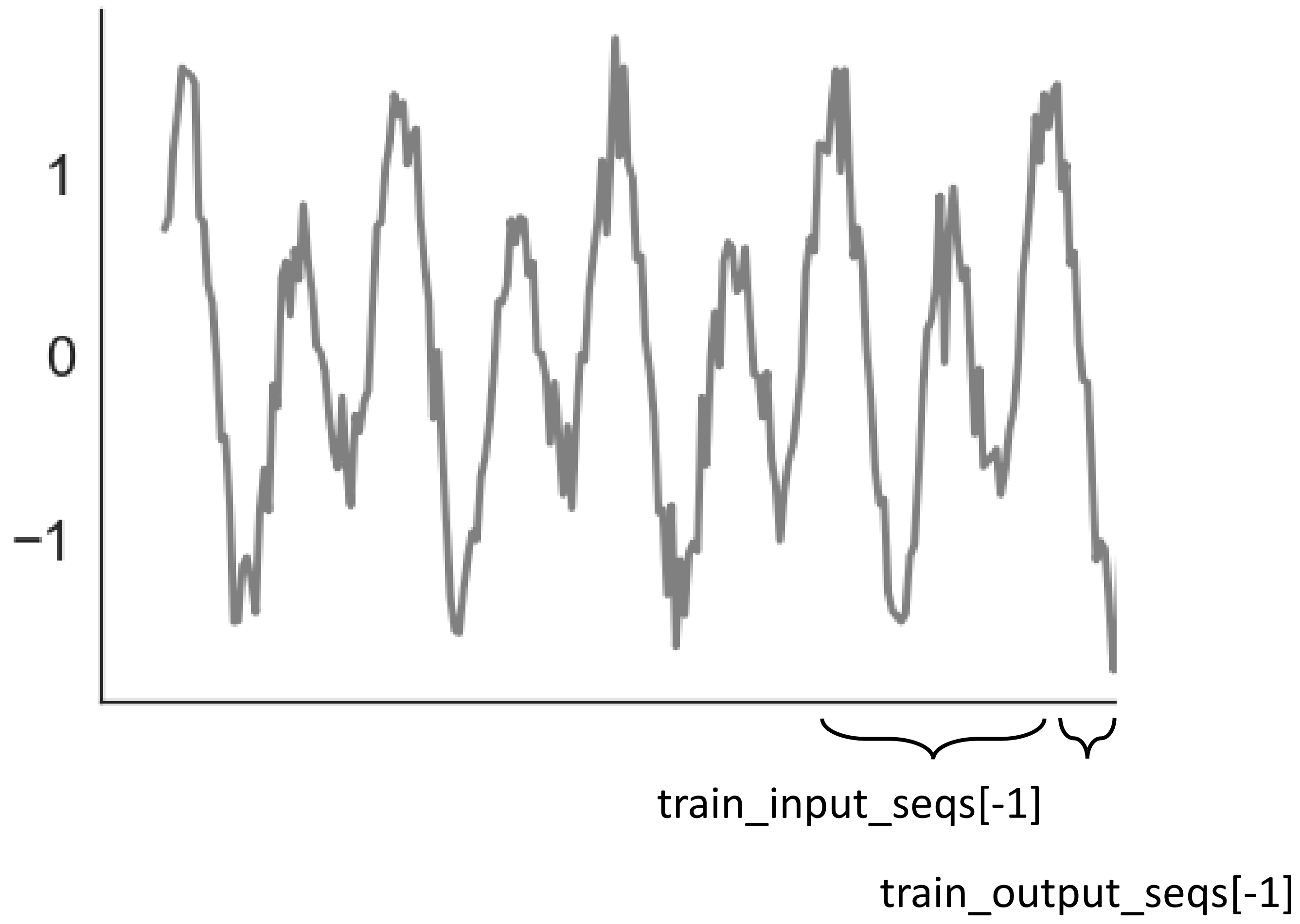
Prepare Data



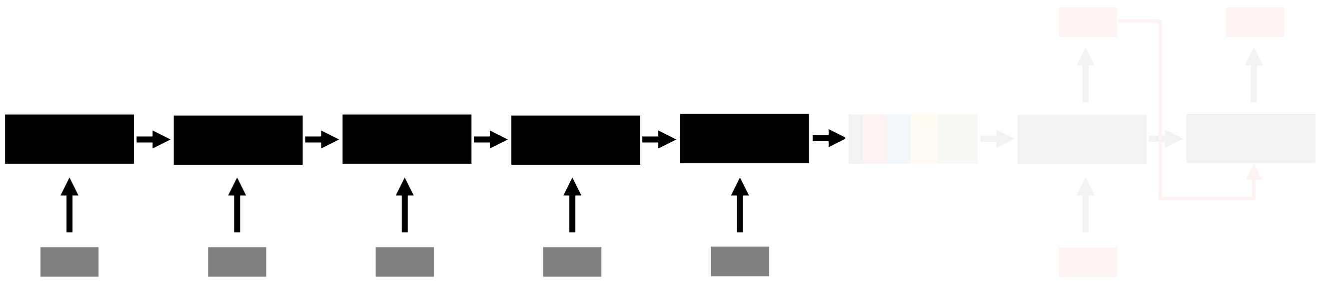
train_input_seqs[2]

train_output_seqs[2]

Prepare Data



Define Model

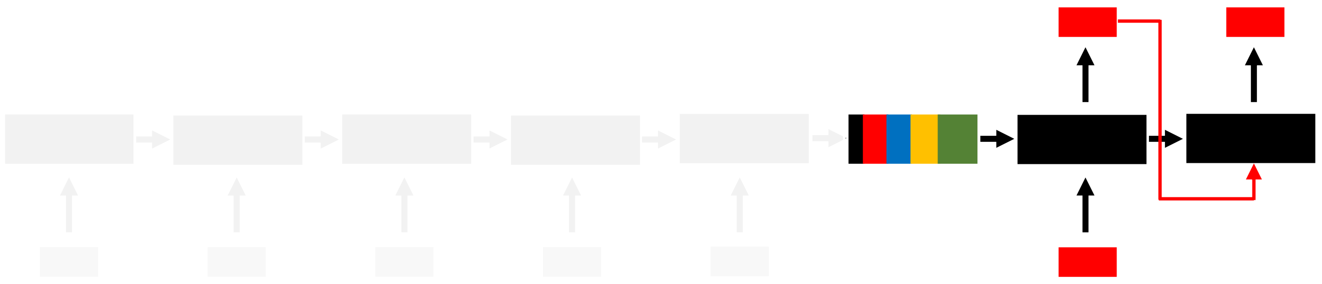


Using LSTM for Encoder

No need for FC layer since encoder only passes hidden states to Decoder

```python
class Encoder(torch.nn.Module):

    def __init__(self, input_size, hidden_size, num_layers):

        super(Encoder, self).__init__()

        self.lstm = torch.nn.LSTM(input_size = input_size, hidden_size = hidden_size,
                                  num_layers = num_layers,
                                  batch_first = True)

    def forward(self, input_seq, hidden_state):

        lstm_out, hidden = self.lstm(input_seq, hidden_state)

        return lstm_out, hidden
```

Define Model

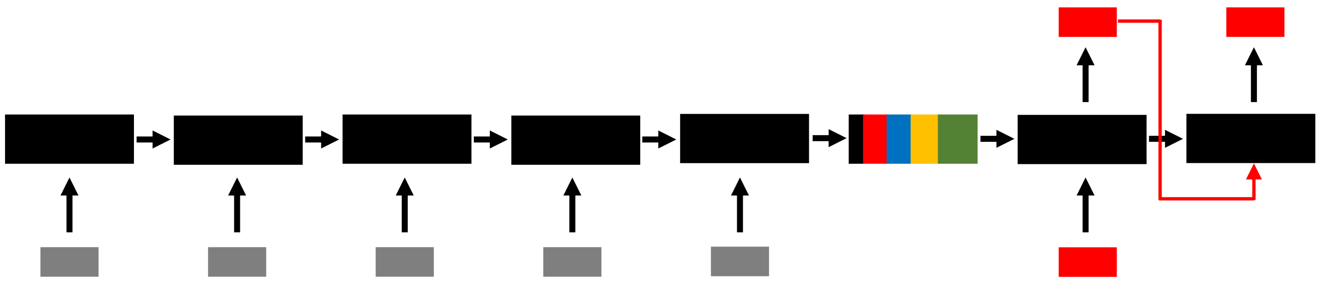


Using LSTM for Decoder

FC layer for converting hidden states to a single number (prediction)

```
class Decoder(torch.nn.Module):

    def __init__(self, input_size, hidden_size, output_size, num_layers):

        super(Decoder, self).__init__()

        self.lstm = torch.nn.LSTM(input_size = input_size, hidden_size = hidden_size,
                                  num_layers = num_layers,
                                  batch_first = True)

        self.fc_decoder = torch.nn.Linear(hidden_size, output_size)

    def forward(self, input_seq, encoder_hidden_states):

        lstm_out, hidden = self.lstm(input_seq, encoder_hidden_states)
        output = self.fc_decoder(lstm_out)

        return output, hidden
```

Define Model



Combine Encoder and Decoder classes into a single class (Encoder_Decoder)

```
class Encoder_Decoder(torch.nn.Module):

    def __init__(self, input_size, hidden_size, decoder_output_size, num_layers):

        super(Encoder_Decoder, self).__init__()

        self.Encoder = Encoder(input_size = input_size, hidden_size = hidden_size,
                               num_layers = num_layers)

        self.Decoder = Decoder(input_size = input_size, hidden_size = hidden_size,
                               output_size = decoder_output_size, num_layers = num_layers)
```

Define Hyperparameters

```python
Encoder_Decoder_RNN = Encoder_Decoder(input_size = 1, hidden_size = 15,
                                      decoder_output_size = 1, num_layers = 1)

learning_rate = 0.01
epochs = 50

batchsize = 5
num_features = train_output_seqs.shape[2]

loss_func = torch.nn.MSELoss()
optimizer = torch.optim.Adam(Encoder_Decoder_RNN.parameters(), lr=learning_rate)

Encoder_Decoder_RNN
```

Define Encoder Decoder Specifics

Define Learning rate, epochs, batchsize and num_features/timestep

Define loss function and optimizer

Identify Tracked Values

```python
train_loss_list = []
```

Empty Python list for keeping track of loss values

Train Model

```
train_input_seqs = torch.from_numpy(train_input_seqs).float()
train_output_seqs = torch.from_numpy(train_output_seqs).float()

train_batches_features = torch.split(train_input_seqs, batchsize)[:-1]
train_batches_targets = torch.split(train_output_seqs, batchsize)[:-1]

batch_split_num = len(train_batches_features)
```

Convert numpy arrays to torch tensors

Split training data into mini-batches (skip last mini-batch since it can have smaller batch size)

Compute total number of mini-batches

Train Model

```
for epoch in range(epochs): # For each epoch

    for k in range(batch_split_num):

        hidden_state = None

        decoder_output_seq = torch.zeros(batchsize, decoder_outputseq_len, num_features)

        optimizer.zero_grad()

        encoder_output, encoder_hidden = Encoder_Decoder_RNN.Encoder(train_batches_features[k], hidden_state)
        decoder_hidden = encoder_hidden

        decoder_input = train_batches_features[k][:, -1, :]
        decoder_input = torch.unsqueeze(decoder_input, 2)

        for t in range(decoder_outputseq_len):

            decoder_output, decoder_hidden = Encoder_Decoder_RNN.Decoder(decoder_input, decoder_hidden)

            decoder_output_seq[:, t, :] = torch.squeeze(decoder_output, 2)

            decoder_input = train_batches_targets[k][:, t, :]
            decoder_input = torch.unsqueeze(decoder_input, 2)

        loss = loss_func(torch.squeeze(decoder_output_seq), torch.squeeze(train_batches_targets[k]))

        train_loss_list.append(loss.item())

        loss.backward()

        optimizer.step()

    print("Averaged Training Loss for Epoch ", epoch,": ", np.mean(train_loss_list[-batch_split_num:]))
```

Define initial hidden states and empty tensor for decoder outputs

Pass training input sequence + hidden states to encoder

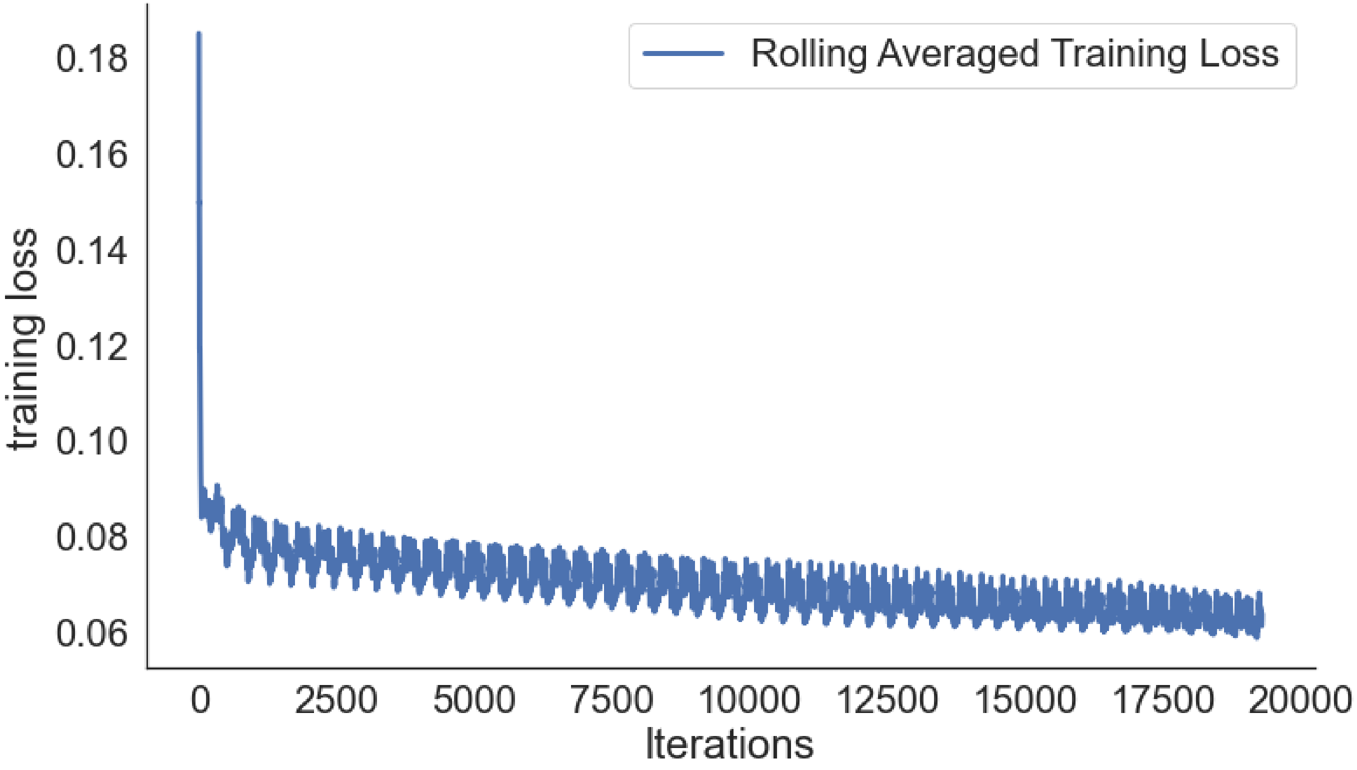
Initial input to decoder = last value of the input sequence

Fill in decoder output tensor by using teacher forcing method
(provide ground truth inputs)

Compute and append Loss
Back-propagation
Update network

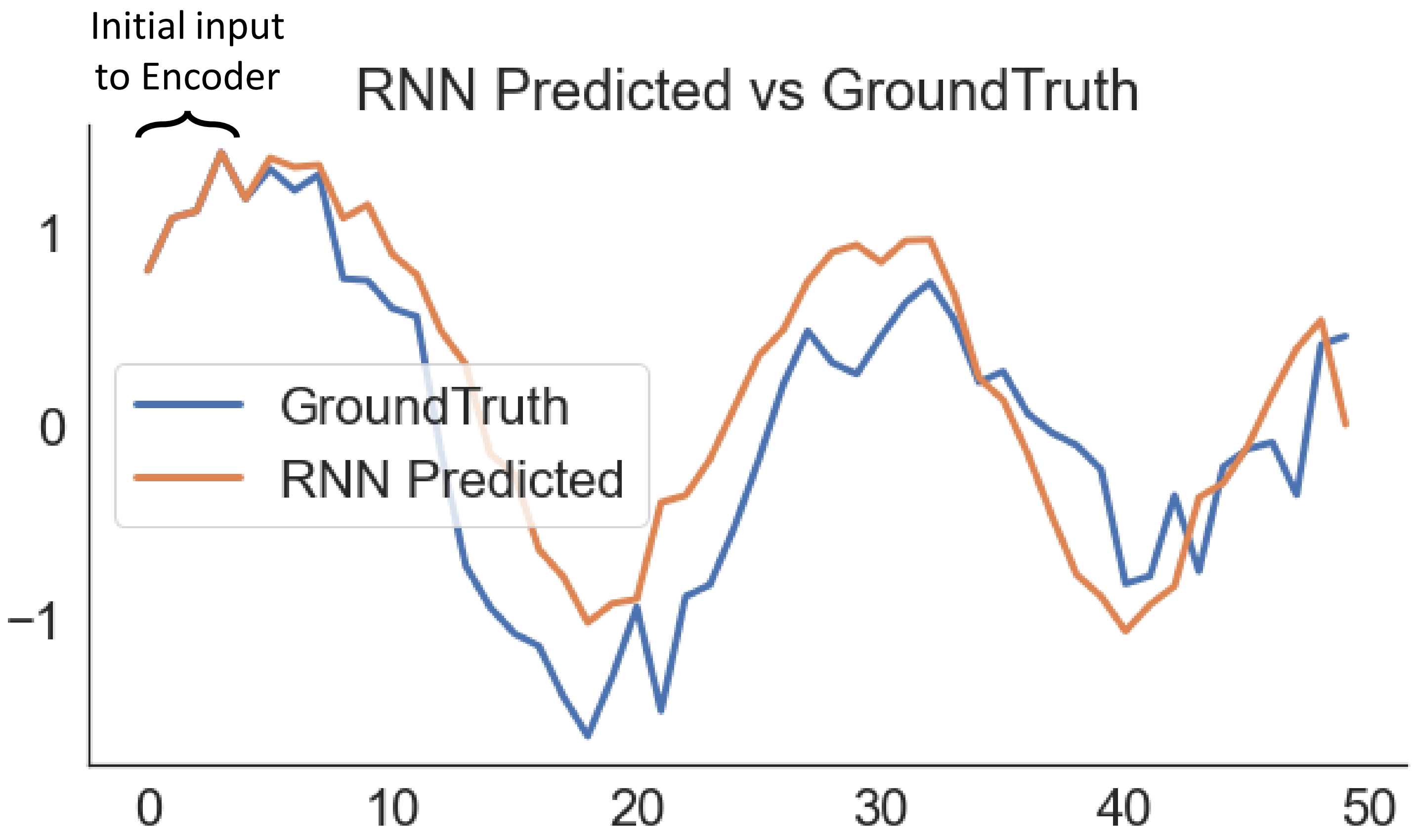
Visualize & Evaluate Model

```
plt.figure(figsize = (12, 7))

plt.plot(np.convolve(train_loss_list, np.ones(100), 'valid') / 100,
         linewidth = 3, label = 'Rolling Averaged Training Loss')
plt.ylabel("training loss")
plt.xlabel("Iterations")
plt.legend()
sns.despine()
```



Plot moving average training loss

Visualize & Evaluate Model



See example notebook for detailed code implementation

LAB 6 ASSIGNMENT:

Stock Prediction AI with Encoder-Decoder RNN

Stock Dataset

	Date	Open	High	Low	Close	Adj Close	Volume
0	2010-06-29	19.000000	25.00	17.540001	23.889999	23.889999	18766300
1	2010-06-30	25.790001	30.42	23.299999	23.830000	23.830000	17187100
2	2010-07-01	25.000000	25.92	20.270000	21.959999	21.959999	8218800
3	2010-07-02	23.000000	23.10	18.709999	19.200001	19.200001	5139800
4	2010-07-06	20.000000	20.00	15.830000	16.110001	16.110001	6866900

- **TSLA.csv**
- 2227 days
- 7 attributes

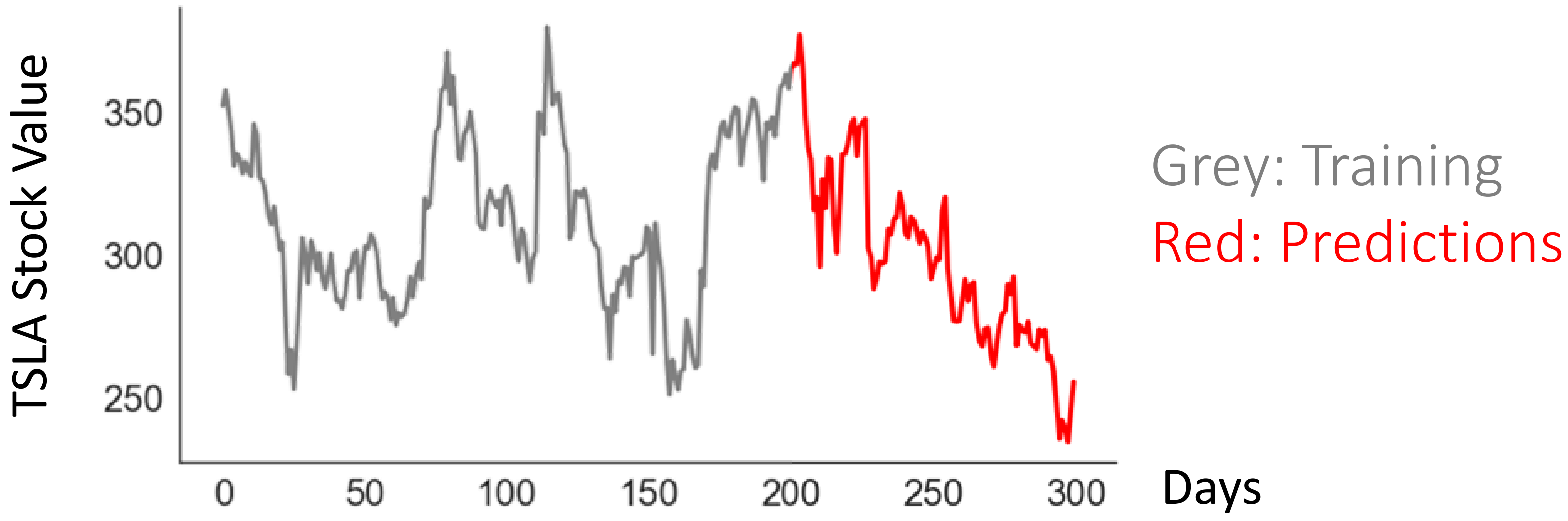
	Date	Open	High	Low	Close	Adj Close	Volume
0	2004-08-19	50.050049	52.082081	48.028027	50.220219	50.220219	44659000
1	2004-08-20	50.555557	54.594593	50.300301	54.209209	54.209209	22834300
2	2004-08-23	55.430431	56.796795	54.579578	54.754753	54.754753	18256100
3	2004-08-24	55.675674	55.855854	51.836838	52.487488	52.487488	15247300
4	2004-08-25	52.532532	54.054054	51.991993	53.053055	53.053055	9188600

- **GOOGL.csv**
- 3702 days
- 7 attributes

	Date	Open	High	Low	Close	Adj Close	Volume
0	1985-01-29	1277.719971	1295.489990	1266.890015	1292.619995	1292.619995	13560000
1	1985-01-30	1297.369995	1305.099976	1278.930054	1287.880005	1287.880005	16820000
2	1985-01-31	1283.239990	1293.400024	1272.640015	1286.770020	1286.770020	14070000
3	1985-02-01	1276.939941	1286.109985	1269.770020	1277.719971	1277.719971	10980000
4	1985-02-04	1272.079956	1294.939941	1268.989990	1290.079956	1290.079956	11630000

- **DJI.csv**
- 8636 days
- 7 attributes

Stock Prediction AI with Encoder-Decoder RNN



In this exercise, you will use Encoder-Decoder RNN architecture to predict the **last 100 days'** stock values.

You are free to pick **one of the stock datasets (TSLA, GOOGL, DJI)** for training and testing your model. Use **closing stock value (i.e., "Close" column)** for both training and testing data.

Feel free to pick **encoder/decoder sequence sizes** of your choice**, LSTM or GRU** for your RNN cell as well as **RNN extensions** such as Deep RNN or Bi-directional RNN.

Before training, **normalize the data** and create **train_input_seqs** and **train_output_seqs** like the example task.

After training, plot your **RNN predicted stock value** against the **ground truth test values** and calculate its **MSE error.** Use **Teacher forcing method** for predicting test outputs.