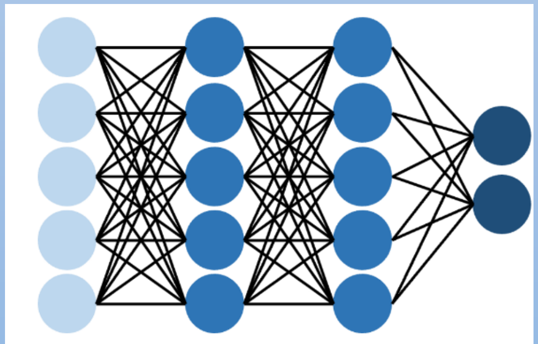
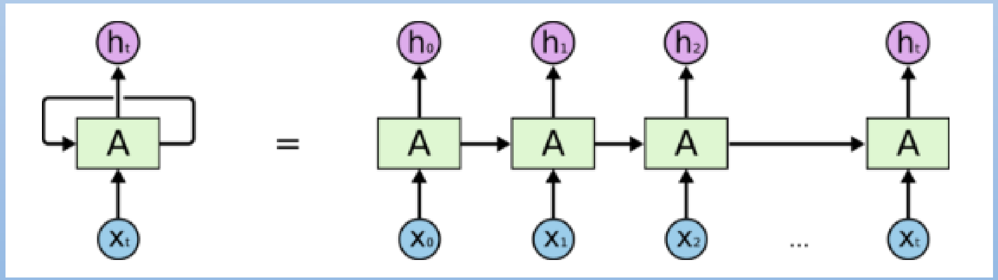
# Module 7 – Part I
# DNN vs RNN for Timeseries
# Intuition

## Standard NN



## Recurrent NN






Jon M. Huntsman School of Business
UtahStateUniversity

Pedram Jahangiry

# Road map!

- Module 1- Introduction to Deep Forecasting
- Module 2- Setting up Deep Forecasting Environment
- Module 3- Exponential Smoothing
- Module 4- ARIMA models
- Module 5- Machine Learning for Time series Forecasting
- Module 6- Deep Neural Networks
- **Module 7- Deep Sequence Modeling (RNN, LSTM)**
- Module 8- Prophet and Neural Prophet

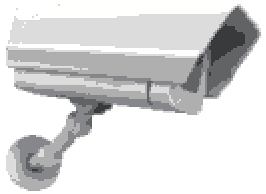
# What is Sequence Data?

- Sequence data refers to any data that has a specific **order** or sequence to it!

Sequence Data

- Time Series Data
  - Regular TS
  - Irregular TS
- Text Data
- Audio Data
- Video Data





Jon M. Huntsman School of Business
UtahStateUniversity

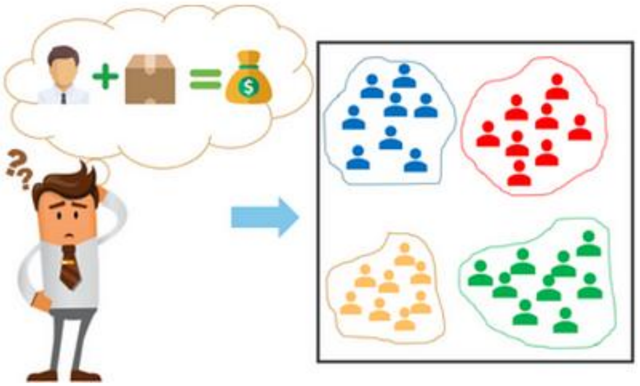
Pedram Jahangiry

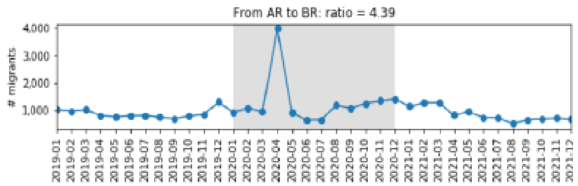
# Time series Tasks

TS tasks

- Forecasting
  - Qualitative
  - Quantitative
- Classification
- Clustering
- Anomaly/Event Detection



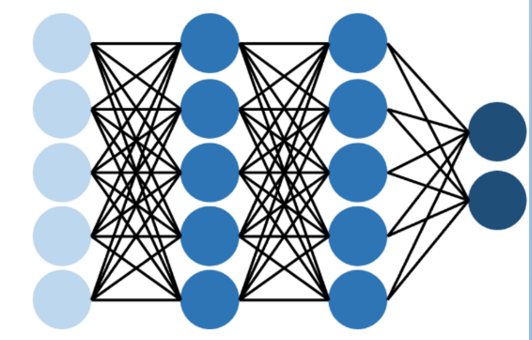




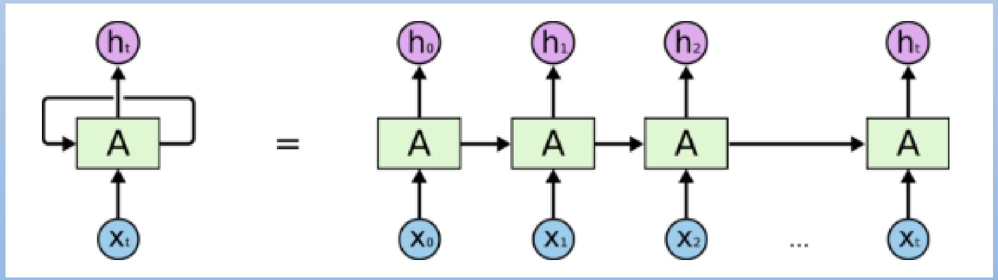

# Understanding DNNs & RNNs for Time Series Forecasting

- Comparing feed-forward networks with sequential models
- Key ideas: data transformation, memory

Standard NN



Recurrent NN




# Feature Engineering in DNN

- Use lagged values (e.g., 12 lags) as independent input features
- Each observation: vector of 12 features representing consecutive time steps

```
Tx = 12 # Number of lags! using the past Tx observations to forecast the next one.
Ty = 1 #  Forecasting Ty outputs at once
X = np.array([series[t:t+Tx] for t in range(len(series) - Tx-Ty+1)])
Y = np.array([series[t+Tx: t+Tx+Ty] for t in range(len(series) - Tx-Ty+1)])
N = len(X)

print("X:", X.shape, "Y:", Y.shape, "N:", N)
```
✓ 0.0s

```
X: (132, 12) Y: (132, 1) N: 132
```

```
Xtrain, Ytrain = X[:-test_period], Y[:-test_period]
Xtest, Ytest = X[-test_period:], Y[-test_period:]

# printing shapes
print(Xtrain.shape, Ytrain.shape, Xtest.shape, Ytest.shape)
```
✓ 0.0s

```
(120, 12) (120, 1) (12, 12) (12, 1)
```

JON M. HUNTSMAN SCHOOL OF BUSINESS
UtahStateUniversity

# It’s All about shapes! DNN

```python
def build_DNN_model(Tx, Ty):
    i = Input(shape=(Tx,))
    x = Dense(32, activation='relu')(i)
    x = Dense(16, activation='relu')(x)
    output = Dense(Ty , activation = 'linear')(x)
    model = Model(i, output)
    model.compile(loss='mse', optimizer='adam')
    return model
```

| Layer (type) | Output Shape | Param # |
|---|---|---|
| input_layer (InputLayer) | (None, 12) | 0 |
| dense (Dense) | (None, 32) | 416 |
| dense_1 (Dense) | (None, 16) | 528 |
| dense_2 (Dense) | (None, 1) | 17 |

# It’s All about shapes! DNN

{

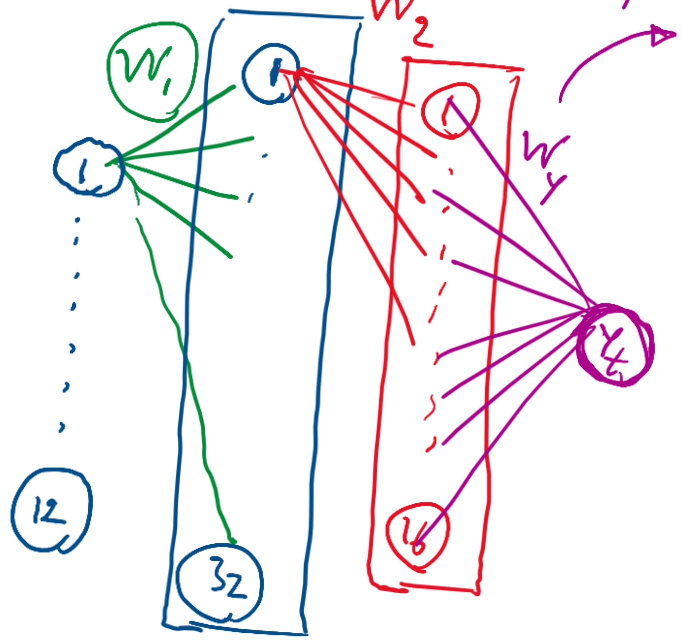
DNN

univariate ts with 1000 hours

$X$: 12 lags

$Y_{t-12} : X_1$

$Y_{t-1} : X_{12}$




$W_y : (16 \times 1)$ $b_y : (1 \times 1)$

$W_2 : 32 \times 16$

$b_2 : 16 \times 1$

$$Y_t = \sigma\left(W_y^T \cdot output_2 + b_y\right)$$

$$\underbrace{(1 \times 16)(16 \times 1) + 1 \times 1}_{(1 \times 1)}$$

$X : (12 \times 1)$

$W_1 : (12 \times 32)$

$b_1 : (32 \times 1)$

$$output_1 = \sigma\left(W_1^T \cdot X + b_1\right) \; : \; 32 \times 1$$

$$\underbrace{(32 \times 12)(12 \times 1) + (32 \times 1)}$$

$$output_2 = \sigma\left(W_2^T \cdot output_1 + b_2\right) \rightarrow 16 \times 1$$

$$(16 \times 32)(32 \times 1) + 16 \times 1$$

Jon M. Huntsman School of Business
Utah State University

Pedram Jahangiry

# Batch Training in DNNs

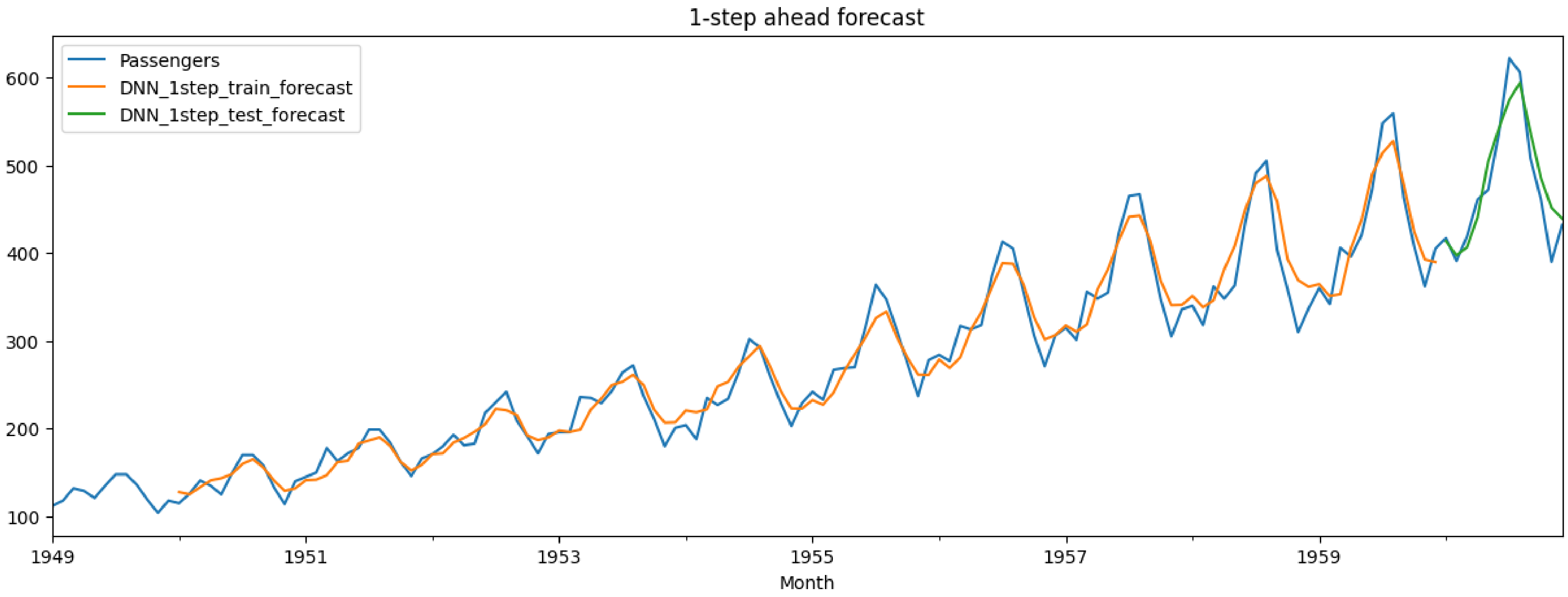
- Data is shuffled into batches (e.g., batch size = 16)
- Temporal order among different observations is lost
- Within each observation, the sequential order of lags is preserved

```
Xtrain, Ytrain = X[:-test_period], Y[:-test_period]
Xtest, Ytest = X[-test_period:], Y[-test_period:]

# printing shapes
print(Xtrain.shape, Ytrain.shape, Xtest.shape, Ytest.sha
✓ 0.0s
(120, 12) (120, 1) (12, 12) (12, 1)
```

```
# training the model
model_DNN.fit(Xtrain, Ytrain, epochs=100, batch_size=16, validation_data=(Xtest, Ytest))

Epoch 1/100
8/8 [==============================] - 1s 32ms/step - loss: 287500.1562 - val_loss: 652289.2500
Epoch 2/100
8/8 [==============================] - 0s 9ms/step - loss: 215688.1094 - val_loss: 487607.0938
Epoch 3/100
8/8 [==============================] - 0s 10ms/step - loss: 161705.9688 - val_loss: 362520.8750
```

# Learning in DNNs

- Model learns to map fixed windows of past values to a target
- Temporal relationships are implicitly modeled through feature patterns



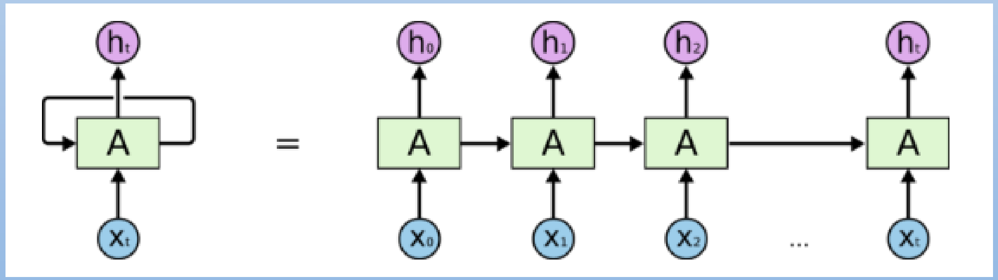

Jon M. Huntsman School of Business
Utah State University

Pedram Jahangiry

# How RNNs Process Time Series Data

- Sequential Processing:
  - Input is the time series itself (one feature)
  - RNN unrolls over a sequence (e.g., sequence length = 12)

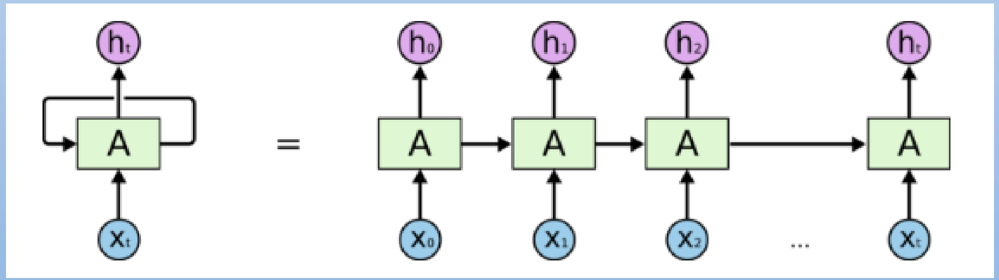
Recurrent NN



# How RNNs Process Time Series Data

- Hidden State Mechanism:
  - Hidden state carries information from previous time steps
  - Explicitly models temporal dependencies

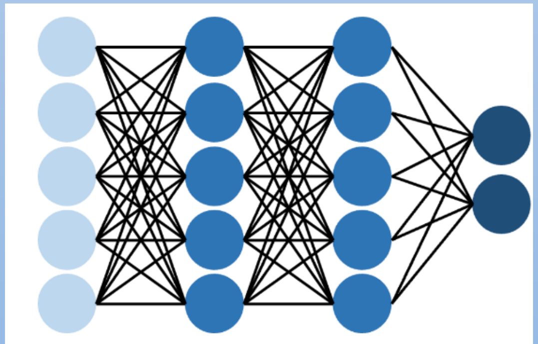
Recurrent NN



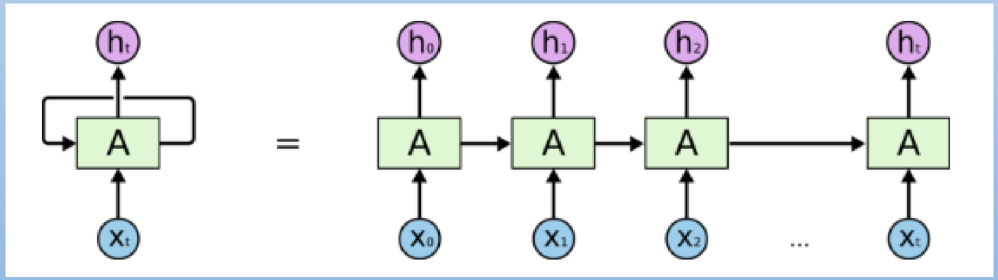
# How RNNs Process Time Series Data

- Key Difference from DNNs:
  - DNNs treat lagged inputs as independent features
  - RNNs connect time steps via hidden states, preserving order

Standard NN



Recurrent NN



# Memory in RNNs

- **Sequence Length:**
  - Sets the potential memory span/length (how many past time steps are seen)
  - Limited by practical issues (e.g., vanishing gradients limit long-term retention)
- **Hidden State Size:**
  - Determines the capacity/depth of the memory
  - Larger hidden states can capture richer, more complex patterns

# Feature Engineering in RNN?

```python
# now let's do a simple RNN model with sequence length of 12
sequence_length = 12
n_features = 1
```

✓ 0.0s

Preparing the data for sequence modeling:

```python
Tx = sequence_length # Number of lags! using the past Tx observations to forecast the next one.
Ty = 1 #  Forecasting Ty outputs at once
X = np.array([series[t:t+Tx] for t in range(len(series) - Tx-Ty+1)])
# we need to reshape X as sequence of data for RNN:
X = np.array(X).reshape(-1, Tx, 1)


Y = np.array([series[t+Tx: t+Tx+Ty] for t in range(len(series) - Tx-Ty+1)])
N = len(X)

print("X:", X.shape, "Y:", Y.shape, "N:", N)
```

✓ 0.0s

```
X: (132, 12, 1) Y: (132, 1) N: 132
```

```python
Xtrain, Ytrain = X[:-test_period], Y[:-test_period]
Xtest, Ytest = X[-test_period:], Y[-test_period:]

# printing shapes
print(Xtrain.shape, Ytrain.shape, Xtest.shape, Ytest.shape)
```

✓ 0.0s

```
(120, 12, 1) (120, 1) (12, 12, 1) (12, 1)
```

# It's All about shapes! RNN

```python
def build_RNN_model(sequence_length, n_features, Ty):
    i = Input(shape=(sequence_length,n_features))
    x = layers.SimpleRNN(16, return_sequences=False)(i)
    # each recurrent cell has one output when return sequence = False
    output = Dense(Ty , activation = 'linear')(x)
    model = Model(i, output)
    model.compile(loss='mse', optimizer='adam')
    return model
```

| Layer (type) | Output Shape | Param # |
|---|---|---|
| input_layer_2 (InputLayer) | (None, 12, 1) | 0 |
| simple_rnn_1 (SimpleRNN) | (None, 16) | 288 |
| dense_4 (Dense) | (None, 1) | 17 |

Jon M. Huntsman School of Business
UtahStateUniversity

Pedram Jahangiry

# All about shapes!

· RNN: Seq-length = 12 → Memory lenght

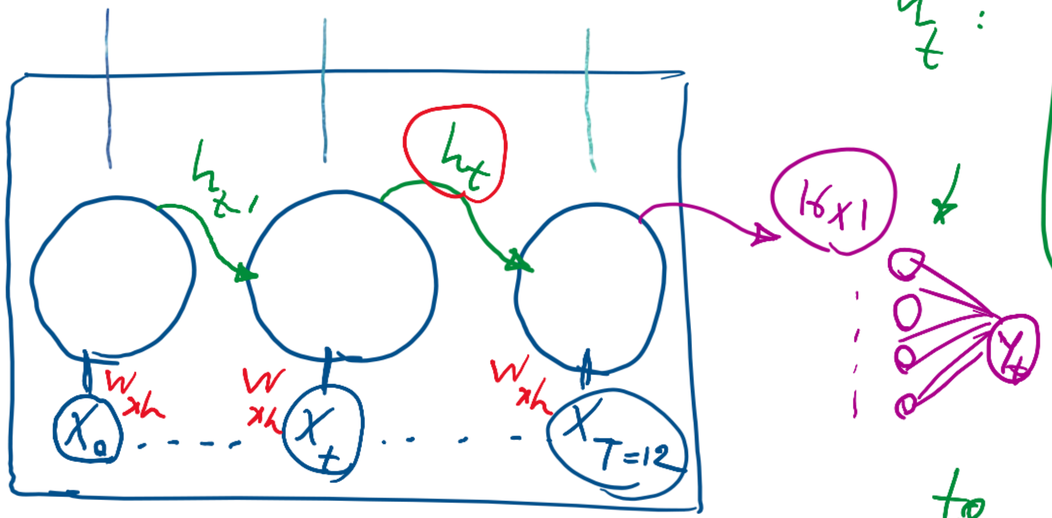
1 SimpleRNN(16) → Memory depth

↓

hidden state dim

$h_t = 16$

RNN_1 : (None, 16 )



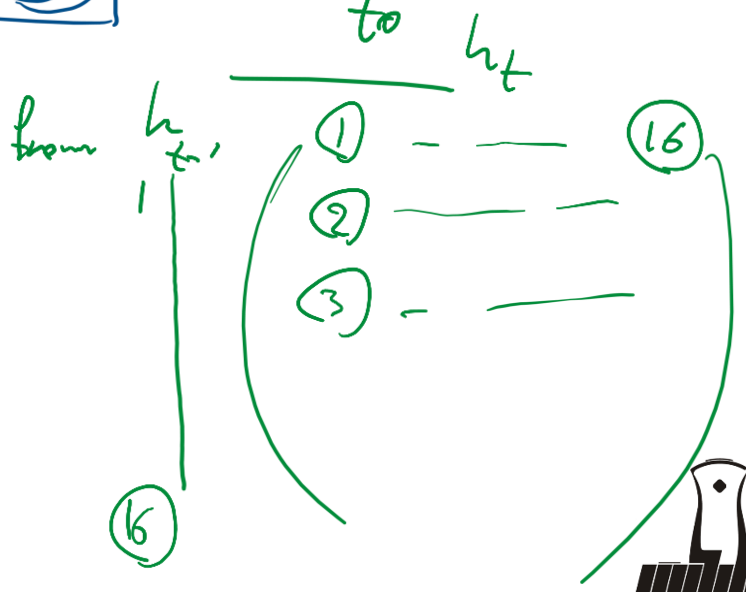

$h_t$ : [1 ⋮ 16] 16x1

$$h_t = \sigma\Big(\underbrace{\underbrace{W_{xh}^T}_{16\times1} \cdot \underbrace{x_t}_{1\times1} + \underbrace{W_{hh}}_{16\times16}\,\underbrace{h_{t-1}}_{16\times1} + \underbrace{b}_{16\times1}}_{16\times1}\Big)$$

RNN_1 : (None, 12, 16)

RNN_2 : (None, 32)

32 → 32x1



# Key Comparisons (DNN vs RNN)

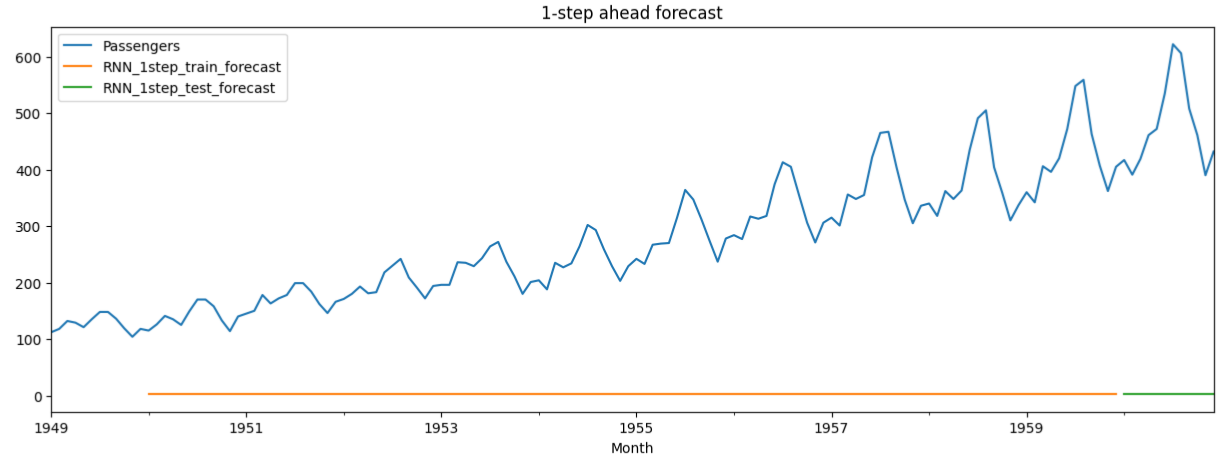
- **DNN:**
  - Simple and effective for short-term dependencies via engineered features
  - Uses engineered lagged features as independent inputs
  - Shuffling within batches loses order between samples

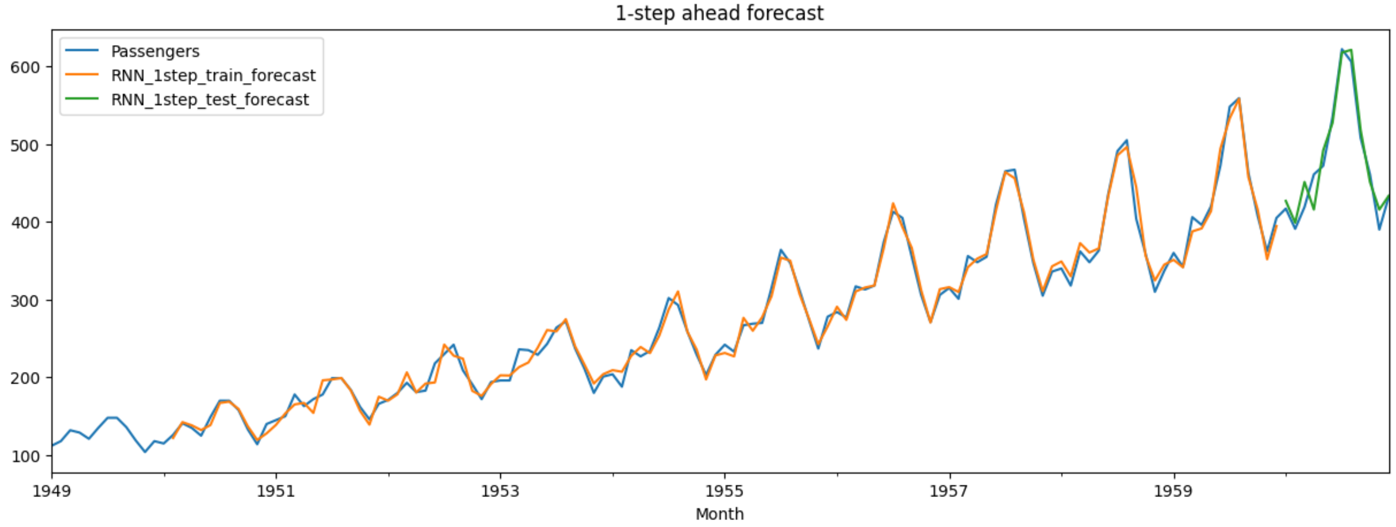
- **RNN:**
  - Designed for sequential data
  - Processes sequences one time step at a time with a hidden state
  - Explicitly captures the order and dependencies in the data
  - Memory is influenced by both sequence length and hidden state size

Jon M. Huntsman School of Business
UtahStateUniversity

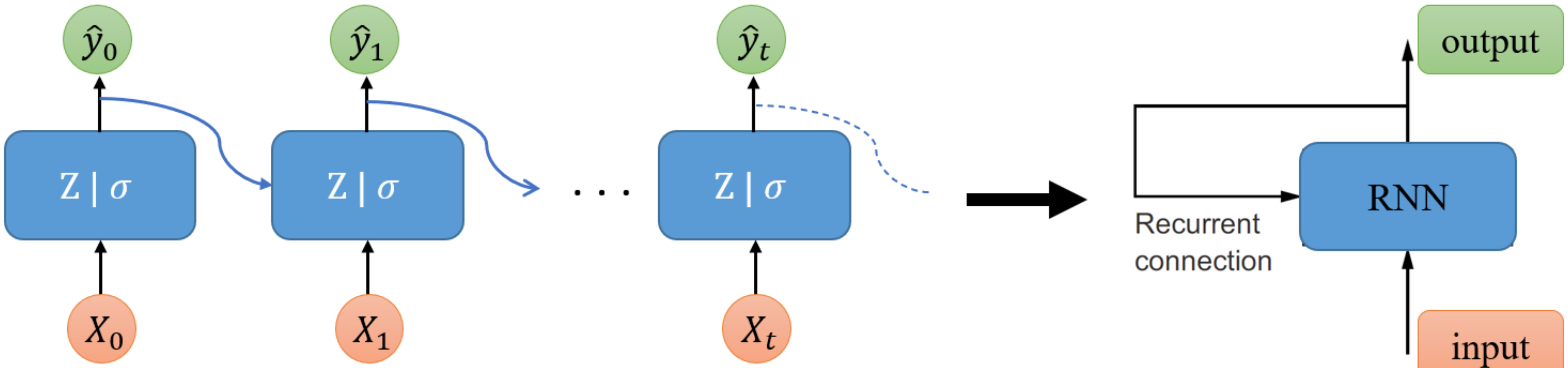
Pedram Jahangiry

# RNN performance (raw data vs pre-processed data)



# Module 7 – Part II
# Deep Sequence Modeling
# Recurrent Neural Networks (RNN)




JON M. HUNTSMAN SCHOOL OF BUSINESS
UtahStateUniversity

Pedram Jahangiry

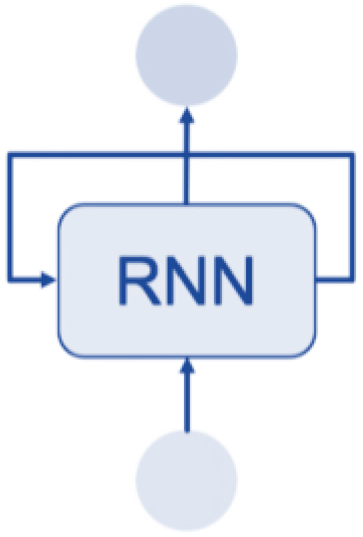
# Sequence Modeling

To model sequence data efficiently, we need a new architecture that:

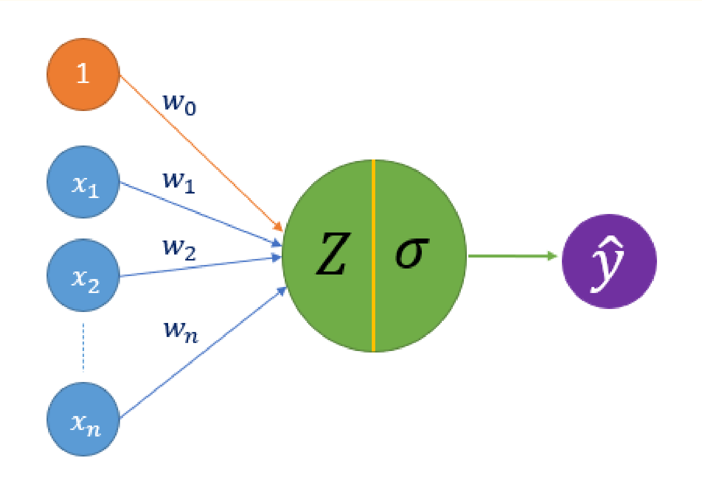
- Preserve the order
- Account for long-term dependencies
- Handle input-length
- Share parameters across the sequence

# What is RNN (Recurrent Neural Network)?

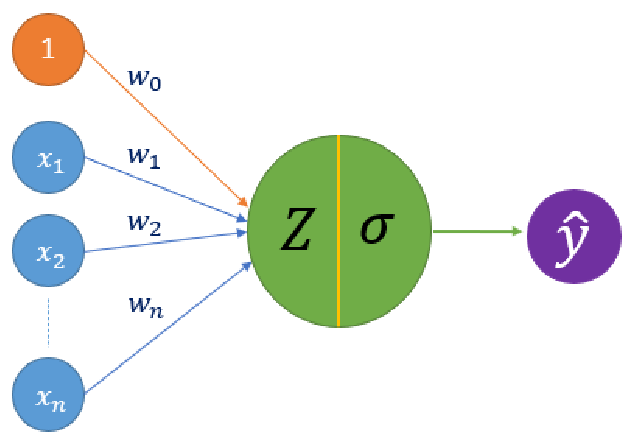
- The architecture of RNNs is inspired by the way biological intelligence processes information incrementally while maintaining an internal model of what it is processing.
- This ability to remember previous inputs and incorporate them into the current output allows RNNs to model sequential data.
- RNN maintains a state that contains information relative to what it has seen so far
- RNNs can be thought of as neural networks with an internal loop, which allows them to process sequences of varying lengths and learn from temporal dependencies.

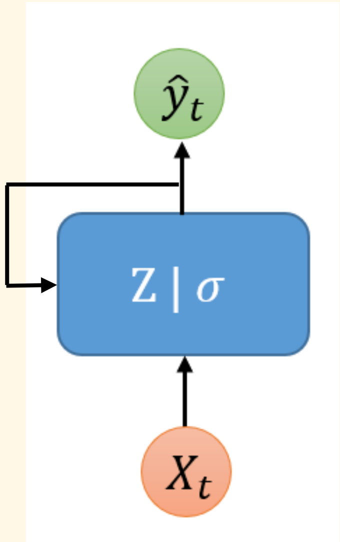


# Perceptron vs Recurrent Cell



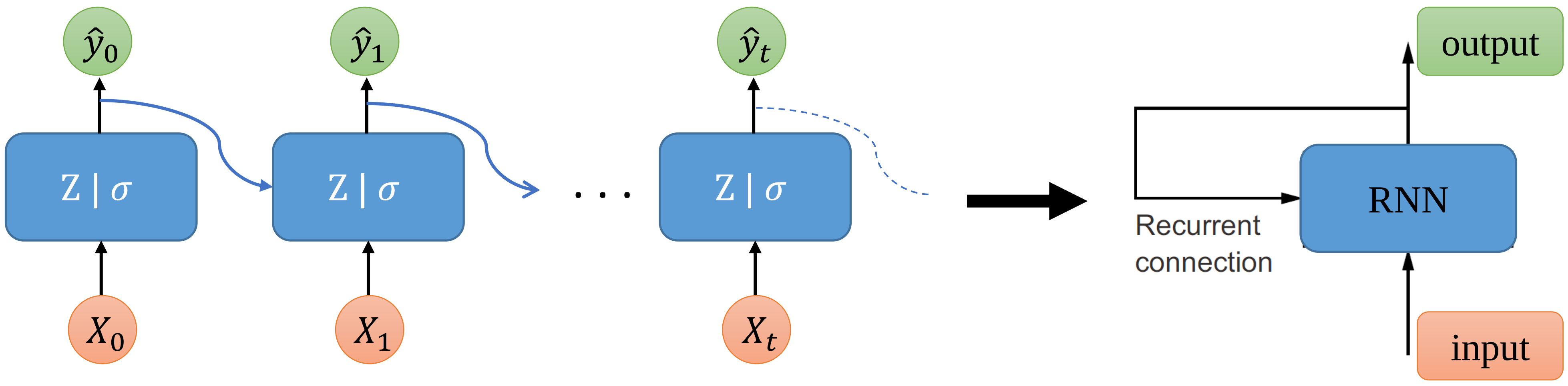

Perceptron



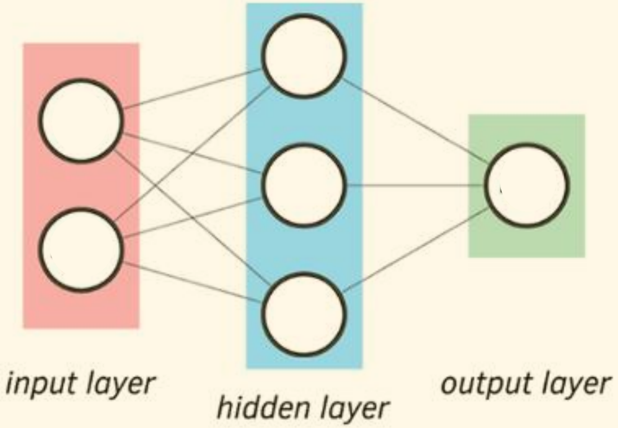



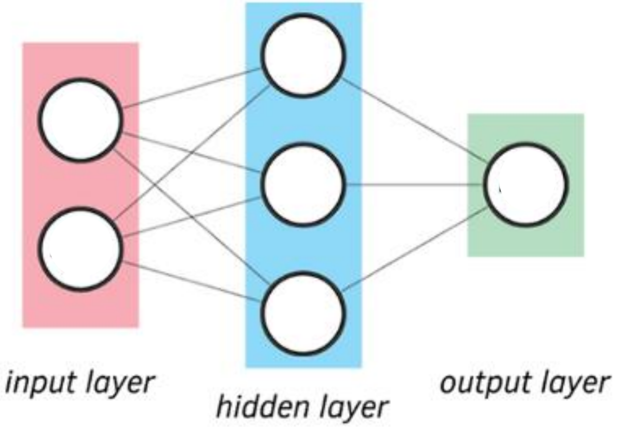

Recurrent Cell

# Unrolling the Recurrent Cell

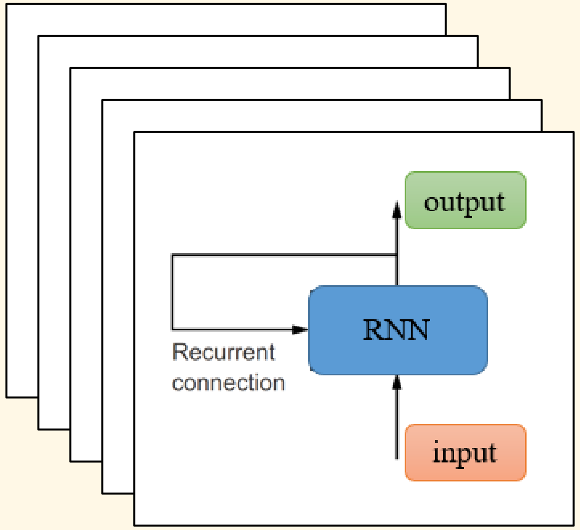


# Dense Layer vs Recurrent Layer



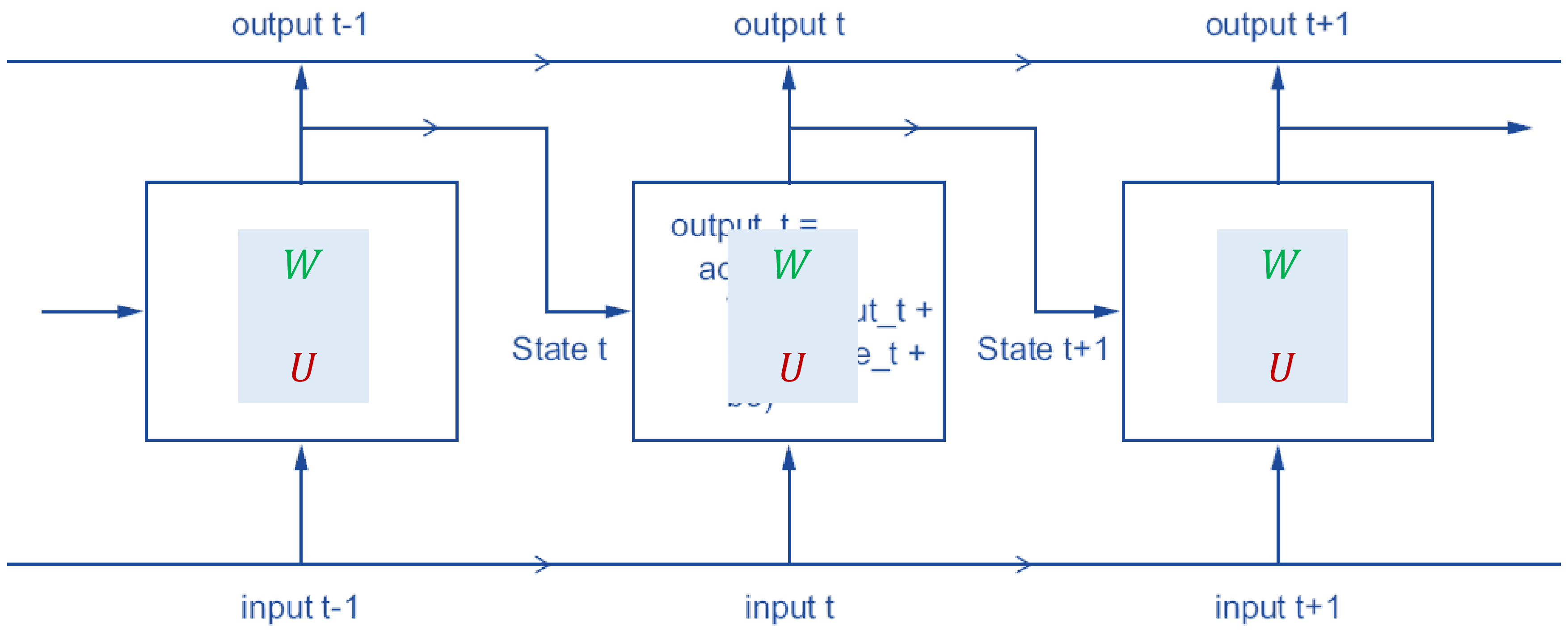

Dense Layer




Recurrent Layer

# Inside the Recurrent Cell

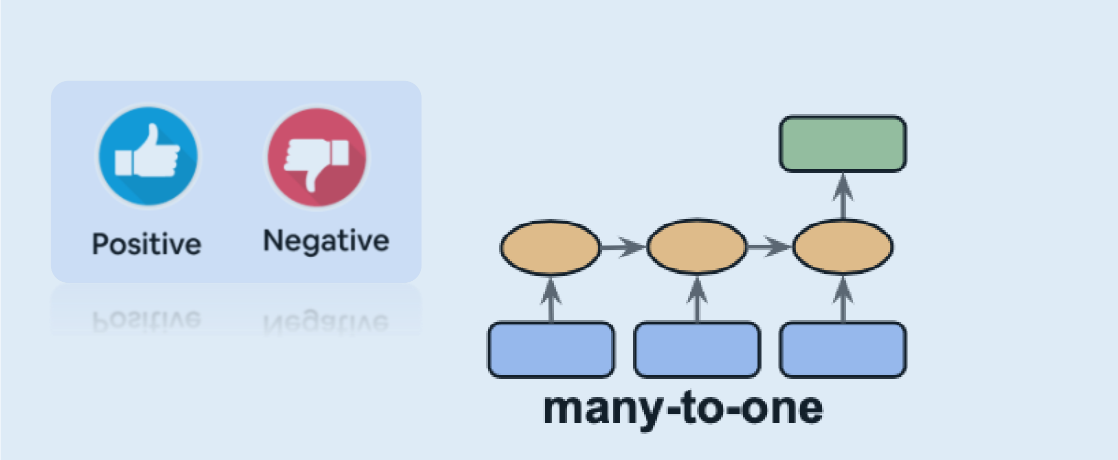
$$output_t = f(input_t, State_t)$$

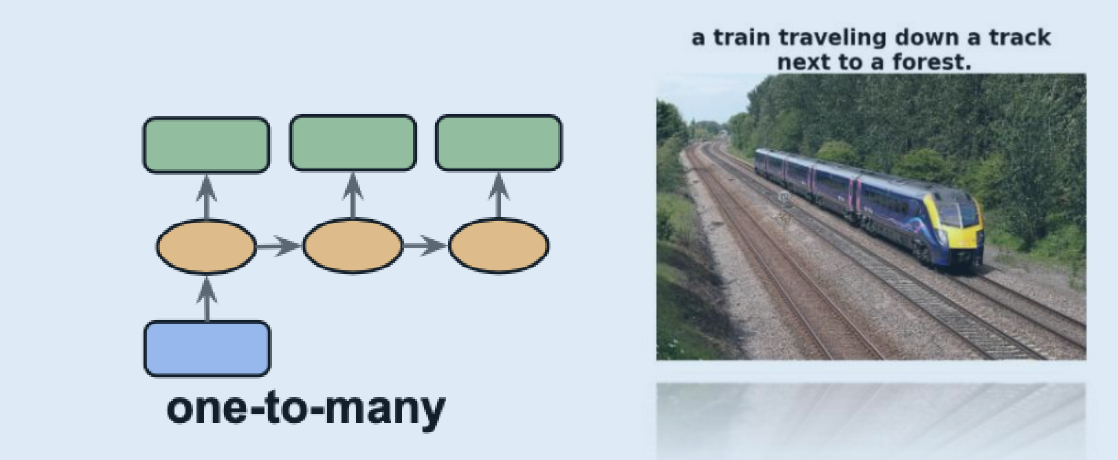


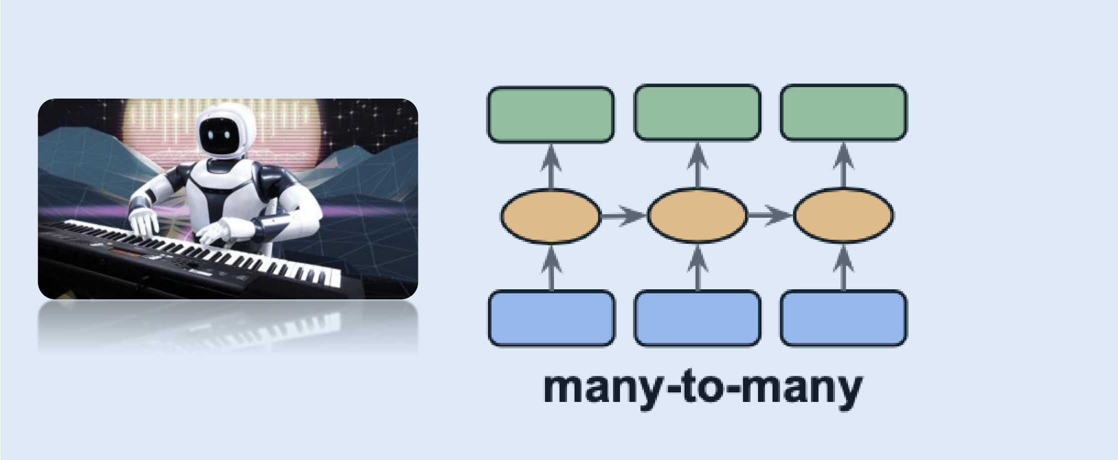

$$s_{t+1} = activation(WX_t + Us_t + b)$$

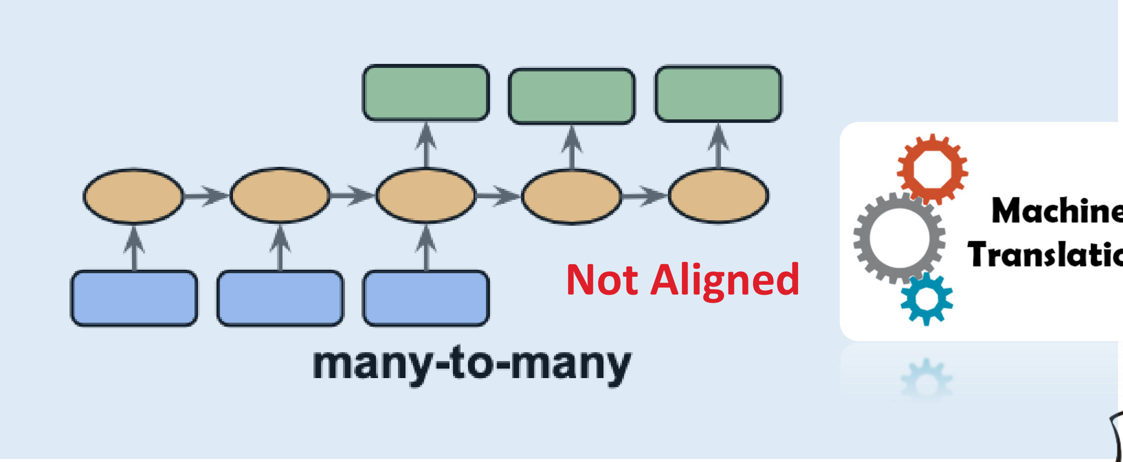
Jon M. Huntsman School of Business
UtahStateUniversity

# RNN architectures



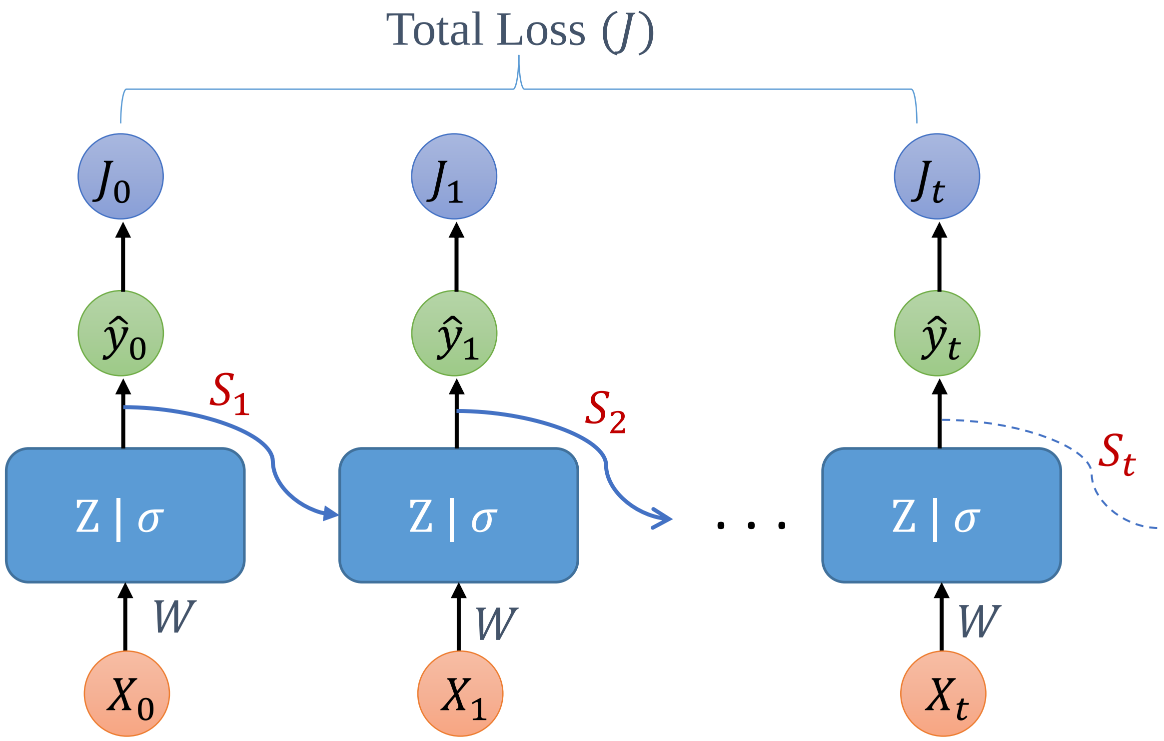



# How does RNN learn representations?

- Backpropagation Through Time (BPTT)
- $\frac{\partial J}{\partial P}$ P are the parameters
- $\frac{\partial J}{\partial W} = \frac{\partial J_0}{\partial W} + \frac{\partial J_1}{\partial W} + ..$
- $\frac{\partial J_0}{\partial W} = \frac{\partial J_0}{\partial y_0}\frac{\partial y_0}{\partial S_0}\frac{\partial S_0}{\partial W}$
- $\frac{\partial J_1}{\partial W} = \frac{\partial J_1}{\partial y_1}\frac{\partial y_1}{\partial S_1}\frac{\partial S_1}{\partial W}$ , $\frac{\partial S_1}{\partial W} = \frac{\partial S_1}{\partial S_0}\frac{\partial S_0}{\partial W}$
- …
- $\frac{\partial J_t}{\partial W} = \sum_{k=0}^{t}\frac{\partial J_t}{\partial y_t}\frac{\partial y_t}{\partial S_t}\frac{\partial S_t}{\partial S_k}\frac{\partial S_k}{\partial W}$

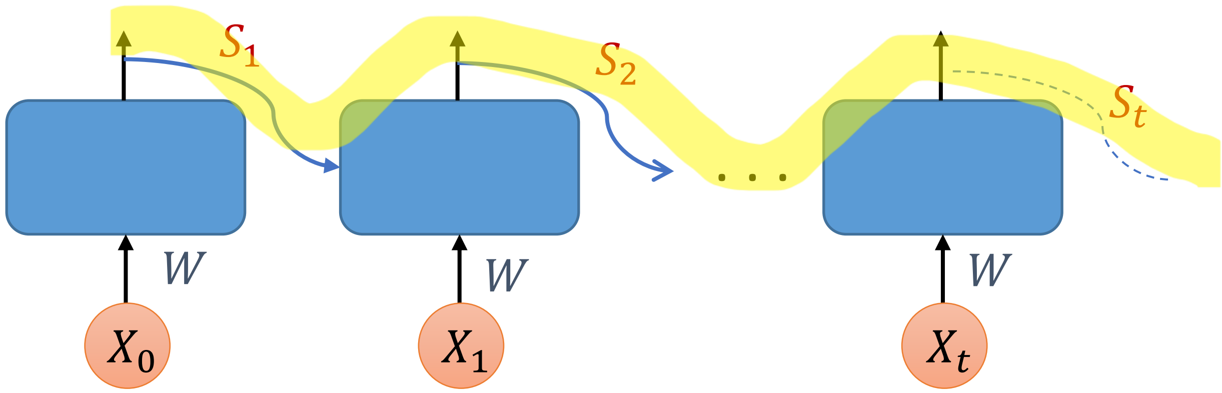


# Vanishing Gradient Problem

- As the time horizon gets bigger, this product gets longer and longer.
- We are multiplying a lot of small numbers → smaller gradients → biased parameters unable to capture long term dependencies.

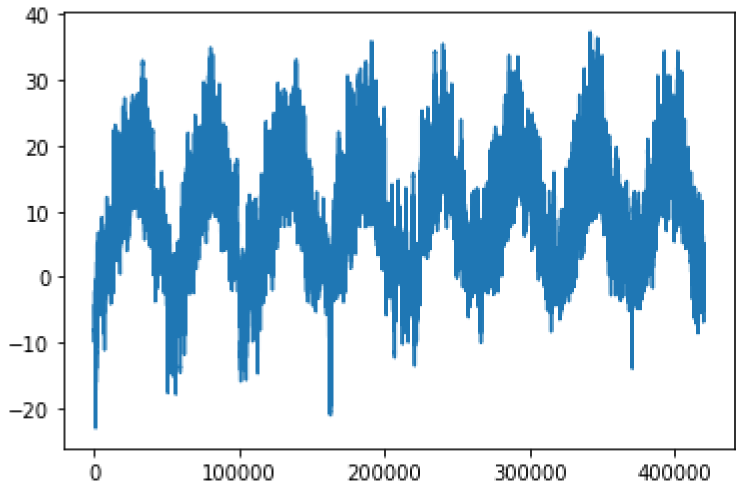
- $\frac{\partial J_t}{\partial W} = \sum_{k=0}^{t} \frac{\partial J_t}{\partial y_t} \frac{\partial y_t}{\partial S_t} \frac{\partial S_t}{\partial S_k} \frac{\partial S_k}{\partial W}$
- $\frac{\partial S_{10}}{\partial S_0} = \frac{\partial S_{10}}{\partial S_9} \frac{\partial S_9}{\partial S_8} \frac{\partial S_8}{\partial S_7} \frac{\partial S_7}{\partial S_6} \cdots \frac{\partial S_1}{\partial S_0}$

$$S_t = activation\left(W X_{t\_1} + U s_{t-1}\right)$$

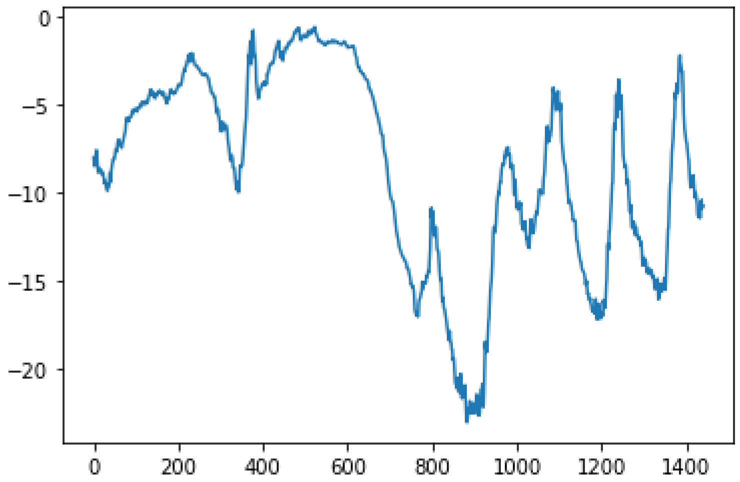


# A simple timeseries with multiple features example

- A temperature forecasting example: [deep-learning-with-python-notebooks]
- Predicting the temperature 24 hours in the future
  - Target: temperature
  - Features: 14 different variables including pressure, humidity, wind direction and etc
  - Data recorded every 10 minutes from 2009-2016




Temperature between 2009-2016




Temperature in the first 10 days: 10*24*6 = 1440

# Preparing the data

- Given the previous 5 days (120 hours) and samples once per hour, can we predict temperature in 24 hours (after the end of the sequence)?
- Data batches:
  - Sequence length = 120
  - [1,2,3,…,120][144]
  - [2,3,4,…,121][145]
  - [3,4,5,…,122][146]
  - Bath size: 256 of these samples are shuffled and batched
  - Sample shape: (256, 120, 14)
  - Target shape: (256,)

# Naïve forecaster: common-sense baseline

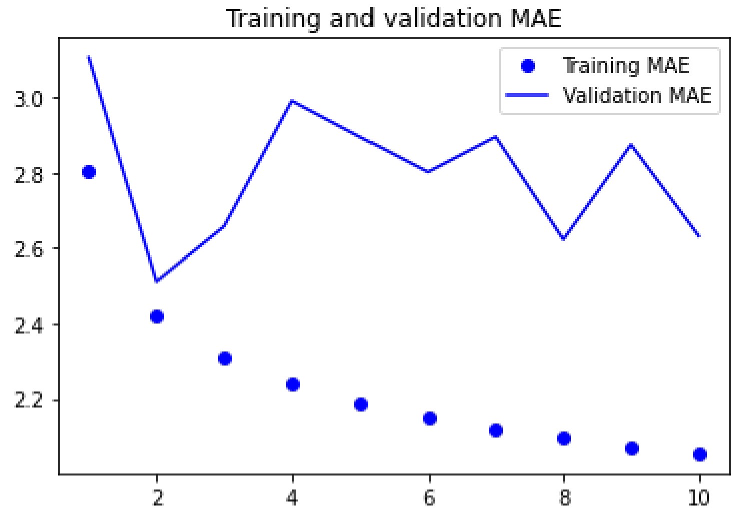
- Temperature 24 hours from now = Temperature right now
- This is our random walk with no drift forecaster.


- Performance:
  - Validation MAE = 2.44 degrees Celsius
  - Test MAE = 2.62 degrees Celsius
  - The baseline model is off by about 2.5 degrees on average. Not bad!!

# Let’s try DNN (Deep Neural Networks)

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Flatten()(inputs)
x = layers.Dense(16, activation="relu")(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)
```

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 120, 14)]         0

 flatten (Flatten)           (None, 1680)              0

 dense (Dense)               (None, 16)                26896

 dense_1 (Dense)             (None, 1)                 17

=================================================================
Total params: 26,913
Trainable params: 26,913
Non-trainable params: 0
_________________________________________________________________
```




- Test MAE = 2.62 degrees Celsius
- No improvement!!
- Flattening a timeseries data is not a good idea!

# Let’s try CNN (Convolutional Neural Networks)

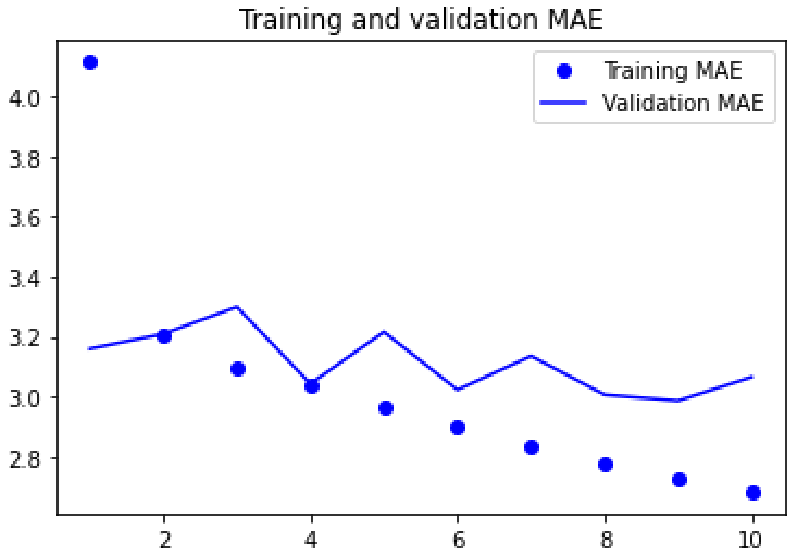
- Motivation: Maybe a temporal convnet could reuse the same representations across different days, much like a spatial convnet can reuse the same representations across different locations in an image!

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Conv1D(8, 24, activation="relu")(inputs)
x = layers.MaxPooling1D(2)(x)
x = layers.Conv1D(8, 12, activation="relu")(x)
x = layers.MaxPooling1D(2)(x)
x = layers.Conv1D(8, 6, activation="relu")(x)
x = layers.GlobalAveragePooling1D()(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)
```

```
_________________________________________________________________
Layer (type)                Output Shape              Param #
=================================================================
input_2 (InputLayer)        [(None, 120, 14)]         0

conv1d (Conv1D)             (None, 97, 8)             2696

max_pooling1d (MaxPooling1D  (None, 48, 8)            0
)

conv1d_1 (Conv1D)           (None, 37, 8)             776

max_pooling1d_1 (MaxPooling  (None, 18, 8)            0
1D)

conv1d_2 (Conv1D)           (None, 13, 8)             392

global_average_pooling1d (G  (None, 8)                0
lobalAveragePooling1D)

dense_2 (Dense)             (None, 1)                 9

=================================================================
Total params: 3,873
Trainable params: 3,873
Non-trainable params: 0
_________________________________________________________________
```

# CNN performance

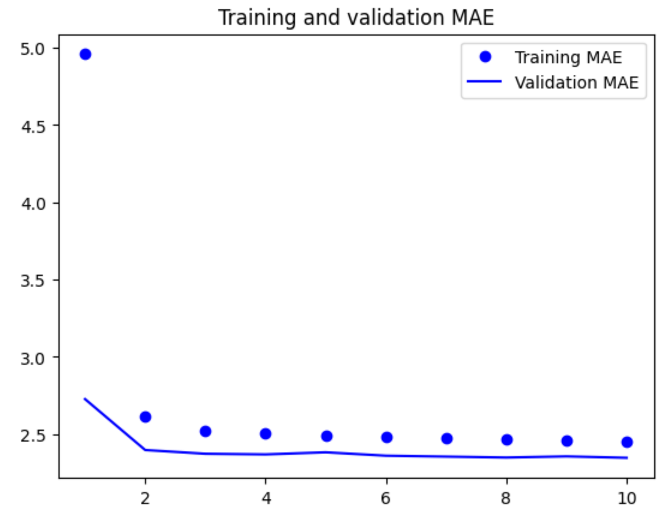
- Test MAE = 3.10 degrees Celsius
- Even worse than the densely connected model!!
  - CNN treats every segment of the data the same way!
  - Pooling layers are destroying order information.



# Let’s try a simple RNN

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.SimpleRNN(16)(inputs)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)
```

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_3 (InputLayer)        [(None, 120, 14)]         0

 simple_rnn (SimpleRNN)      (None, 16)                496

 dense_3 (Dense)             (None, 1)                 17

=================================================================
Total params: 513 (2.00 KB)
Trainable params: 513 (2.00 KB)
Non-trainable params: 0 (0.00 Byte)
```

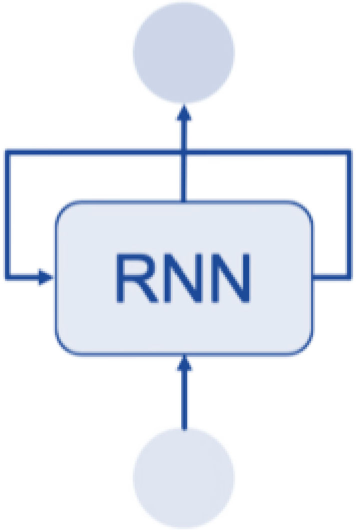



- Baseline Test MAE = 2.62
- Simple RNN Test MAE = 2.51
- beats the naïve forecaster.

# Beyond RNN

RNN can handle the following sequence modeling criteria:

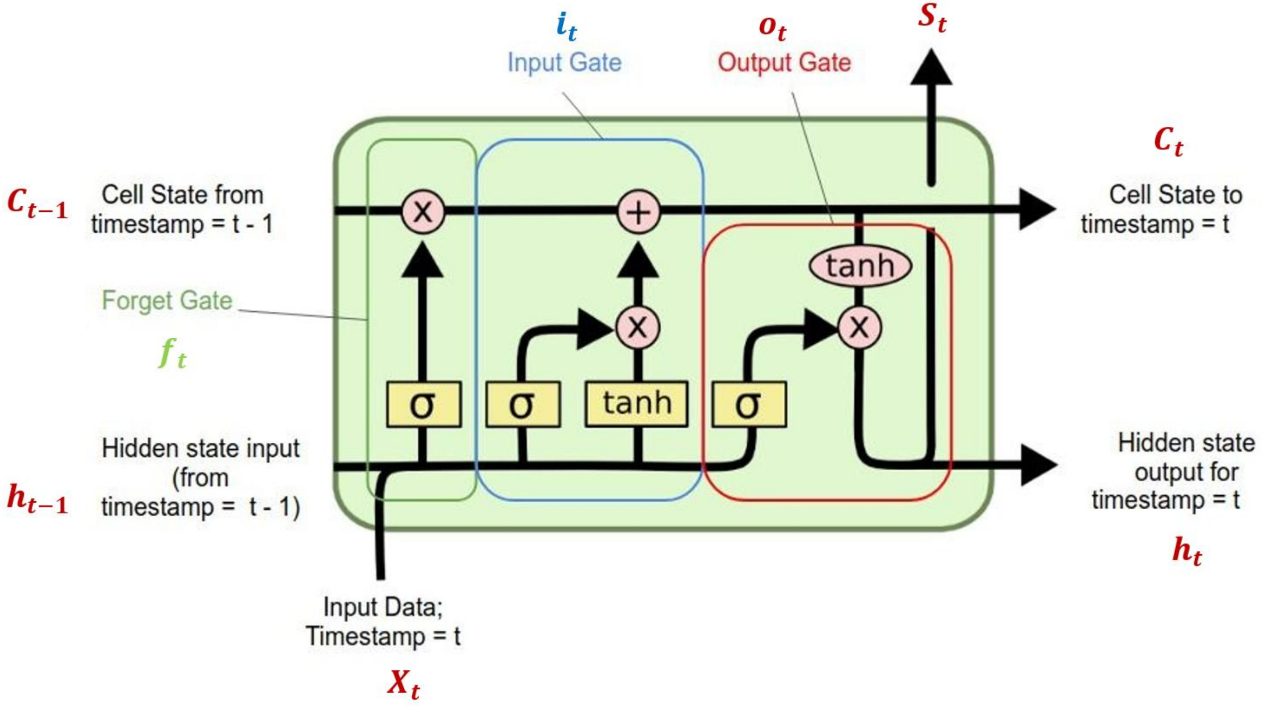
- Preserve the order
- Handle input-length
- Share parameters across the sequence




RNN limitations:

- Does not account for long-term dependencies (only remember short term history )
- Vanishing Gradient Problem

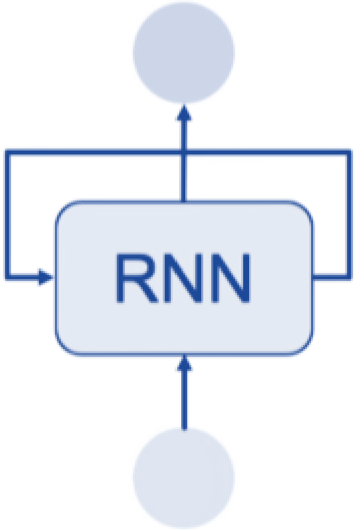
# Module 7 – Part III
# Deep Sequence Modeling
# (Gated cells, LSTM)



# Beyond RNN

RNN can handle the following sequence modeling criteria:

- Preserve the order
- Handle input-length
- Share parameters across the sequence

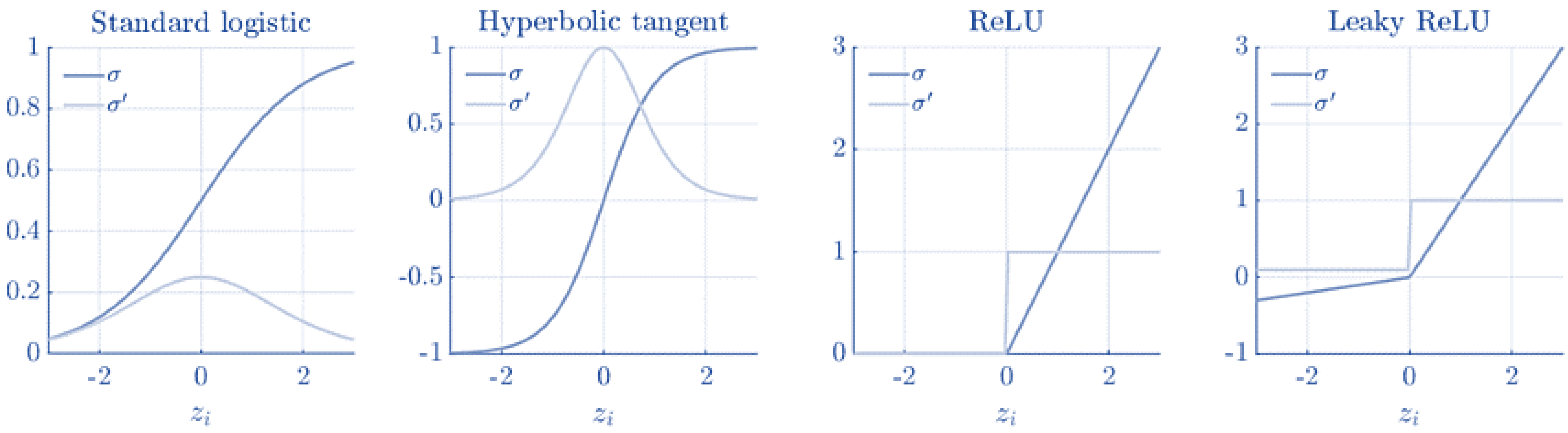



RNN limitations:

- Does not account for long-term dependencies (only remember short term history )
- Vanishing Gradient Problem

# How to solve vanishing gradient problem

1. Use Activation Function that prevents fast shrinkage of gradient



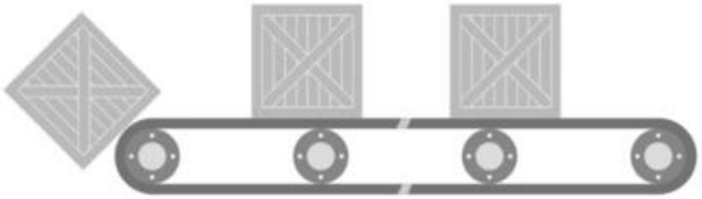

$$S_t = activation(WX_{t-1} + Us_{t-1})$$

# How to solve vanishing gradient problem

1. Use Activation Function that prevents fast shrinkage of gradient
2. Use weight initialization techniques that ensure that the initial weights are not too small
3. Use gradient clipping which limits the magnitude of the gradients from becoming too small (vanishing gradient) or too large (exploding gradient)
4. Use batch normalization, which normalizes the input to each layer and helps to reduce the range of activation values and thus the likelihood of vanishing gradients.
5. Use a different optimization algorithm that is more resilient to vanishing gradients, such as Adam or RMSprop.
6. **Gated cells:** Use some sort of **skip connections**, which allow gradients to bypass some of the layers in the network and thus prevent them from becoming too small.

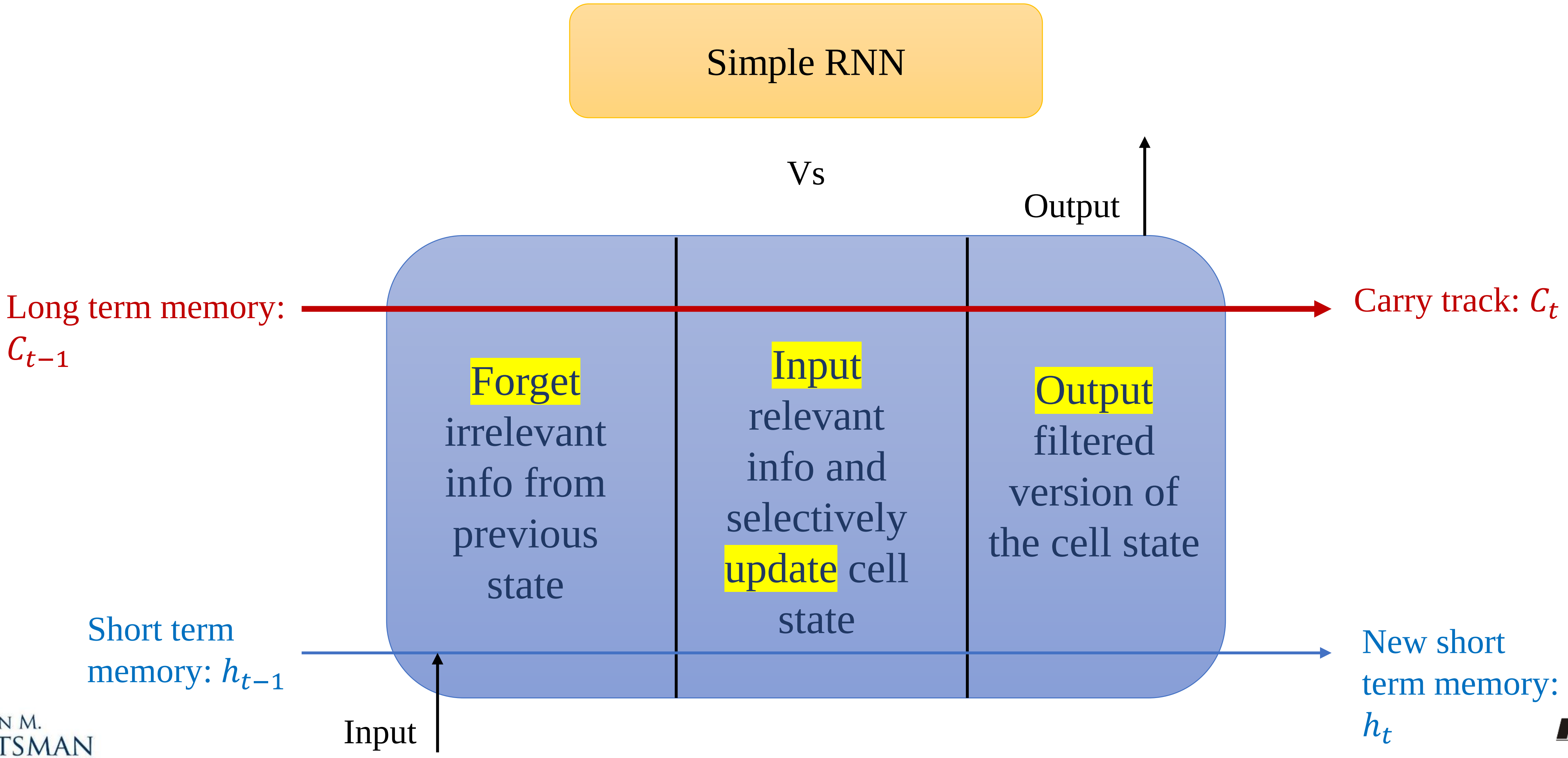
# Gated cells

- Instead of using a simple RNN cell, let's use a more complex cell with gates which control the flow of information.
- Think of a conveyer belt running parallel to the sequence being processed:
  - Information can jump on → transported to a later timestep → jump off when needed.
  - This is what a gated cell does! Analogous to residual connections we saw before.

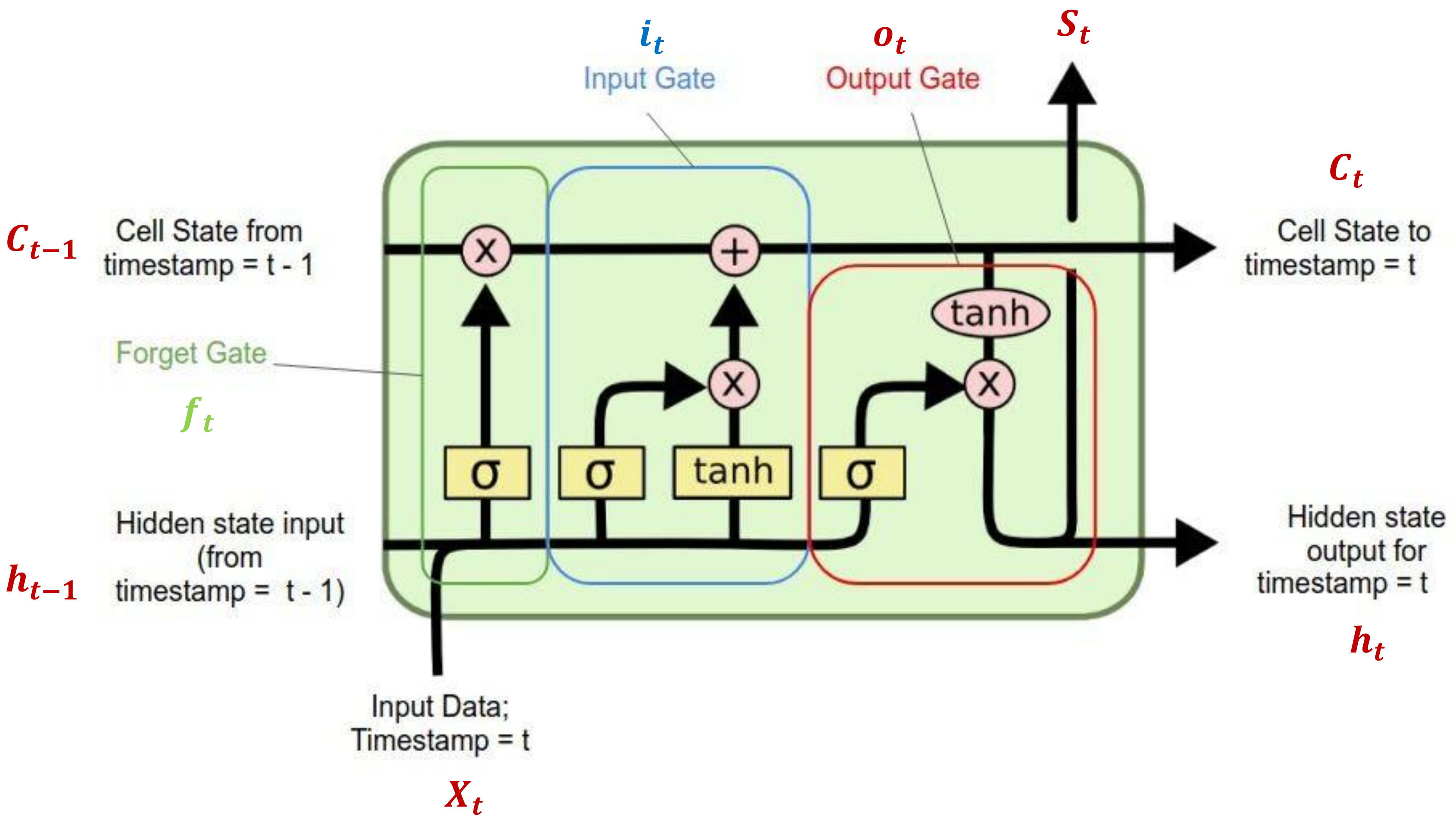


- Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) are two examples of gated cells that can keep track of information throughout many timesteps.

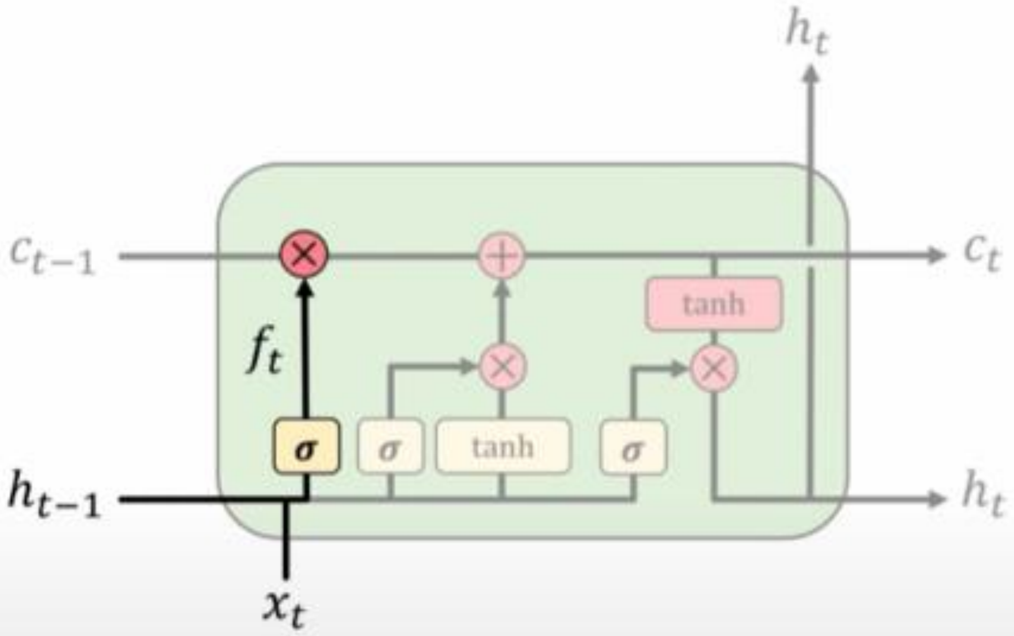
# Inside the LSTM cell



# LSTM details



# LSTMs: forget irrelevant information




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

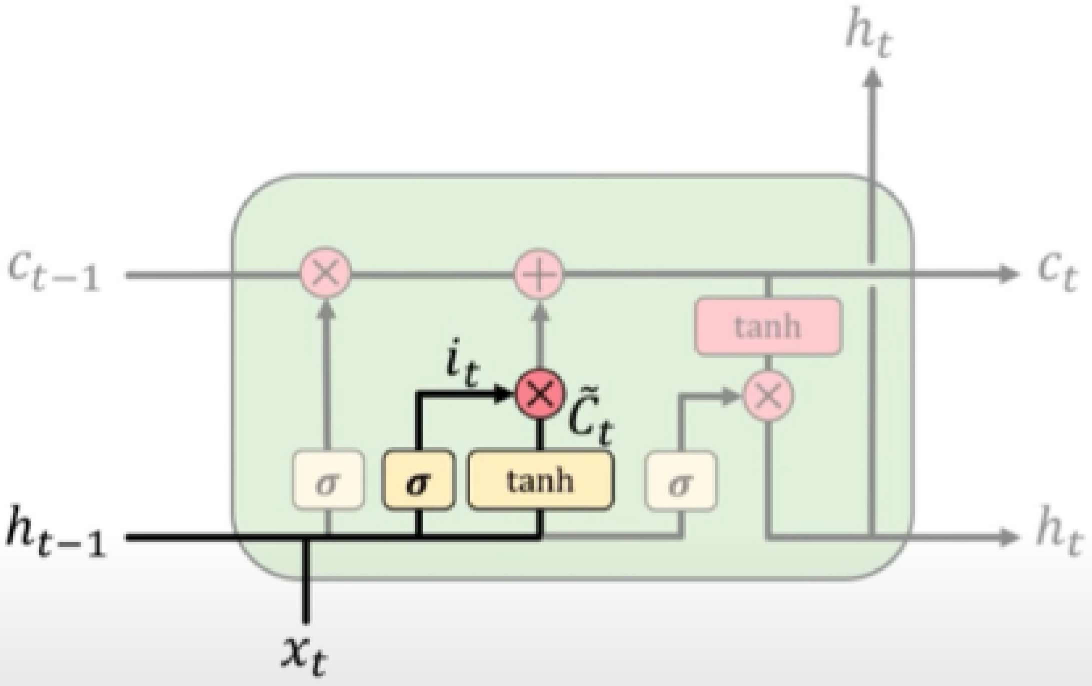
- Use previous cell output and input
- Sigmoid: value 0 and 1 – "completely forget" vs. "completely keep"

*ex: Forget the gender pronoun of previous subject in sentence.*

JON M. HUNTSMAN SCHOOL OF BUSINESS
UtahStateUniversity

Pedram Jahangiry

# LSTMs: identify new information to be stored



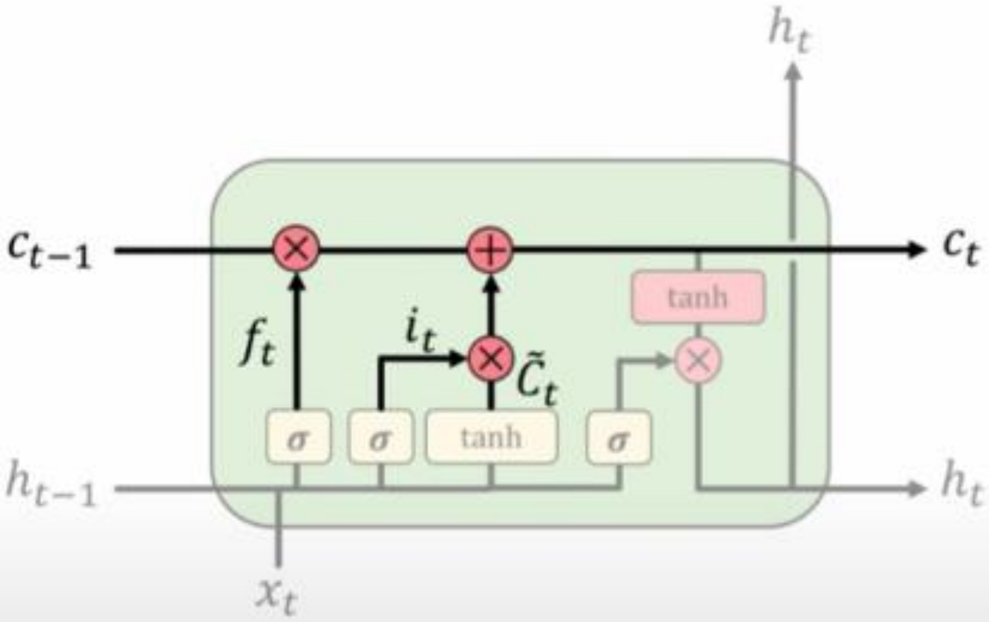

$$i_t = \sigma(\boldsymbol{W_i}[\, h_{t-1}, x_t \,] + b_i)$$

$$\tilde{C}_t = \tanh(\boldsymbol{W_C}[\, h_{t-1}, x_t \,] + b_C)$$

- Sigmoid layer: decide what values to update
- Tanh layer: generate new vector of "candidate values" that could be added to the state

*ex: Add gender of new subject to replace that of old subject.*

# LSTMs: update cell state




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

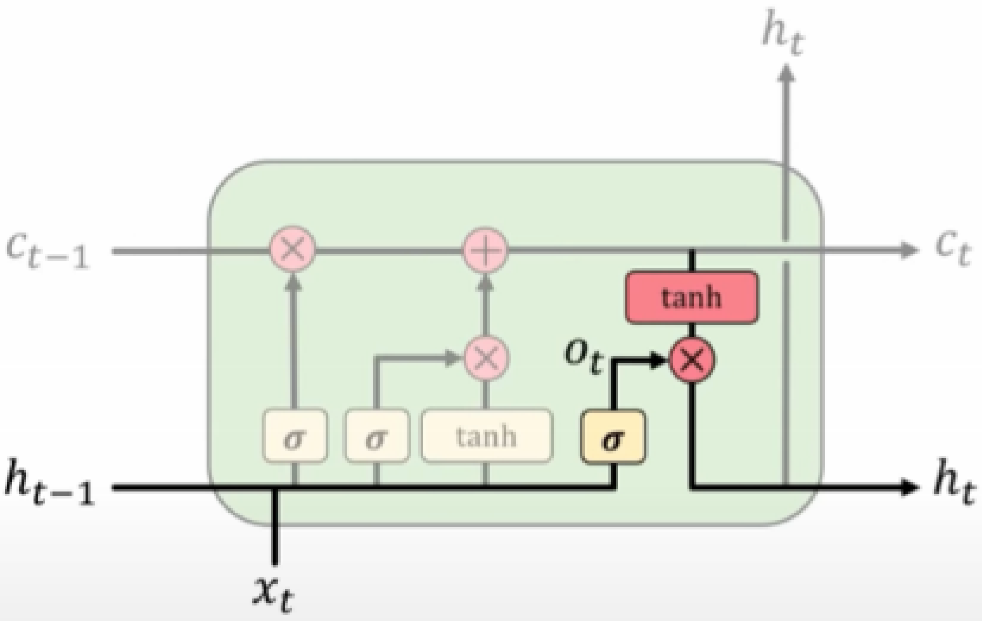
- Apply forget operation to previous internal cell state: $f_t * C_{t-1}$
- Add new candidate values, scaled by how much we decided to update: $i_t * \tilde{C}_t$

*ex: Actually drop old information and add new information about subject's gender.*

Jon M. Huntsman School of Business Utah State University

Pedram Jahangiry

# LSTMs: output filtered version of cell state




$$o_t = \sigma(W_o[\,h_{t-1}, x_t\,] + b_o)$$

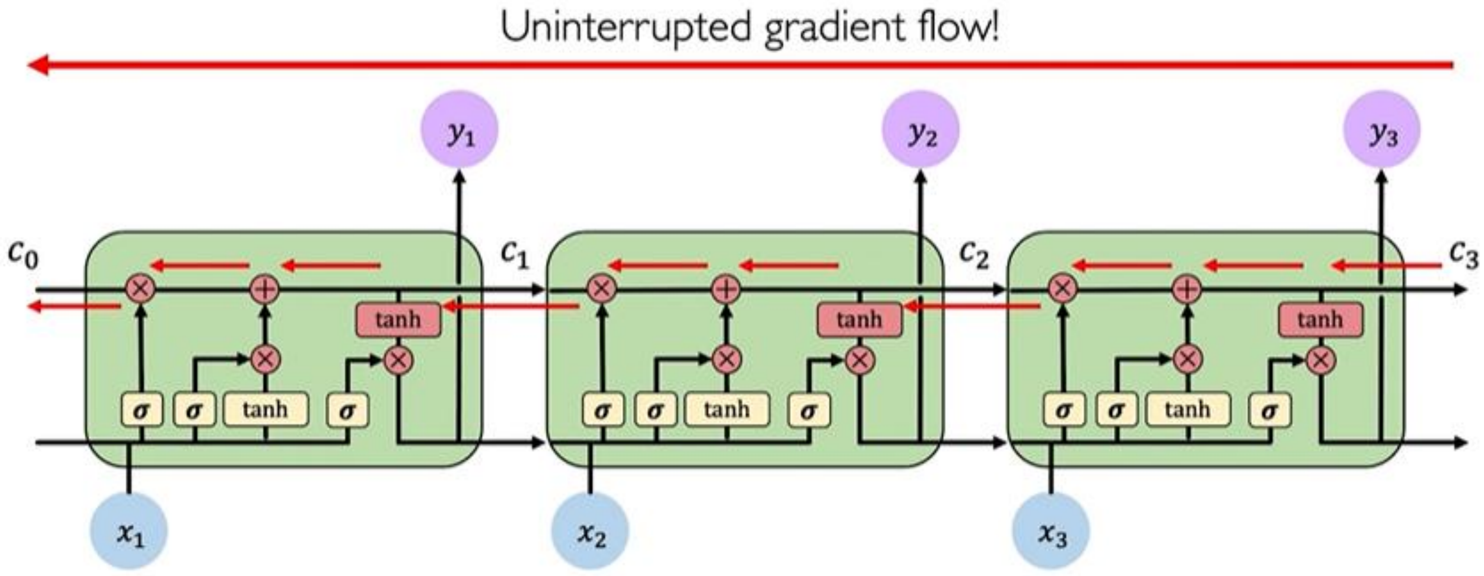
$$h_t = o_t * \tanh(C_t)$$

- Sigmoid layer: decide what parts of state to output
- Tanh layer: squash values between -1 and 1
- $o_t * \tanh(C_t)$: output filtered version of cell state

*ex: Having seen a subject, may output information relating to a verb.*

# LSTM takeaway

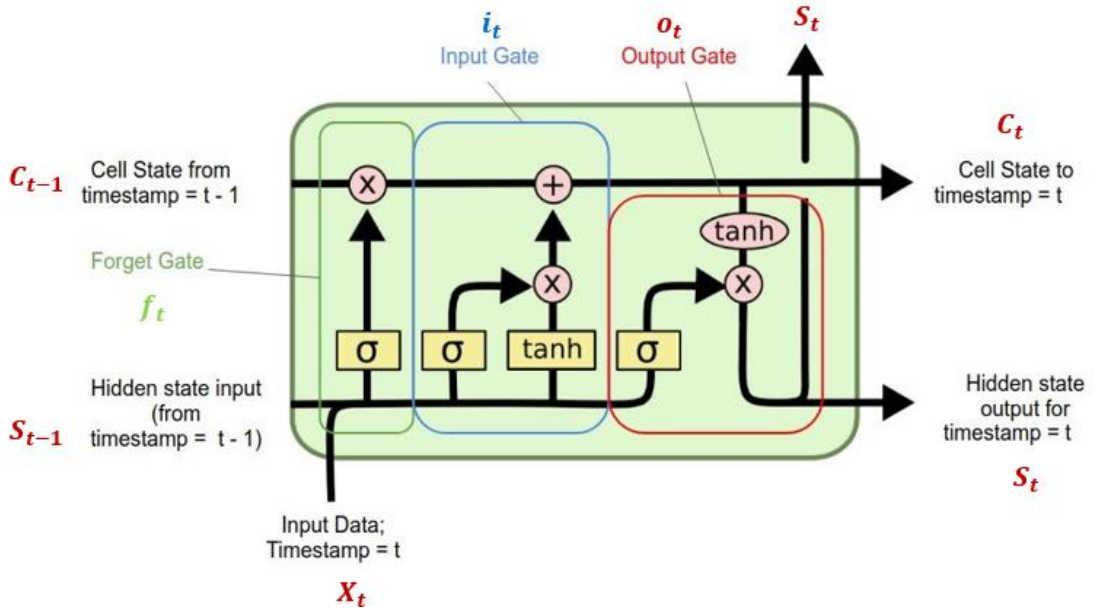
- LSTM uses gates to regulate the information flow (allows past information to be reinjected later)
- This new cell state (carry) can better capture longer term dependencies
- LSTM fights the vanishing gradient problem



# Let's try LSTM on the temperature example

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.LSTM(16)(inputs)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)
```

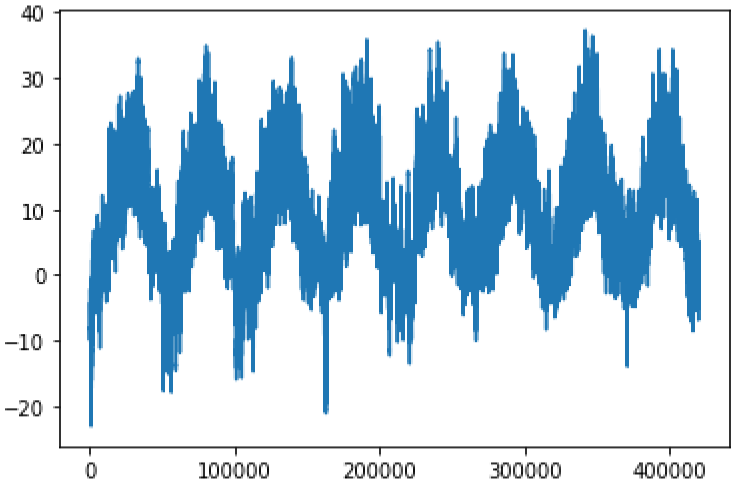
```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_3 (InputLayer)        [(None, 120, 14)]         0

 lstm (LSTM)                 (None, 16)                1984

 dense_3 (Dense)             (None, 1)                 17

=================================================================
Total params: 2,001
Trainable params: 2,001
Non-trainable params: 0
_________________________________________________________________
```



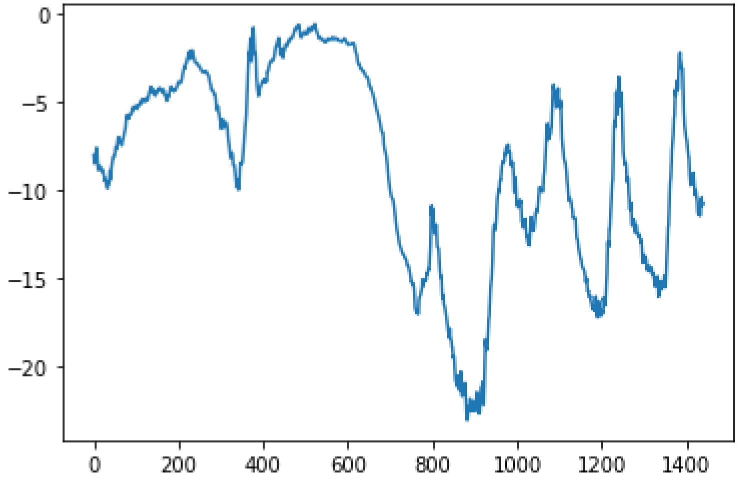

```
output_t = activation(dot(state_t, Uo) + dot(input_t, Wo) + dot(c_t, Vo) + bo)
i_t = activation(dot(state_t, Ui) + dot(input_t, Wi) + bi)
f_t = activation(dot(state_t, Uf) + dot(input_t, Wf) + bf)
k_t = activation(dot(state_t, Uk) + dot(input_t, Wk) + bk)
```

# A simple timeseries with multiple features example

- A temperature forecasting example: [deep-learning-with-python-notebooks]()
- Predicting the temperature 24 hours in the future
  - Target: temperature
  - Features: 14 different variables including pressure, humidity, wind direction and etc
  - Data recorded every 10 minutes from 2009-2016



Temperature between 2009-2016



Temperature in the first 10 days: 10*24*6 = 1440

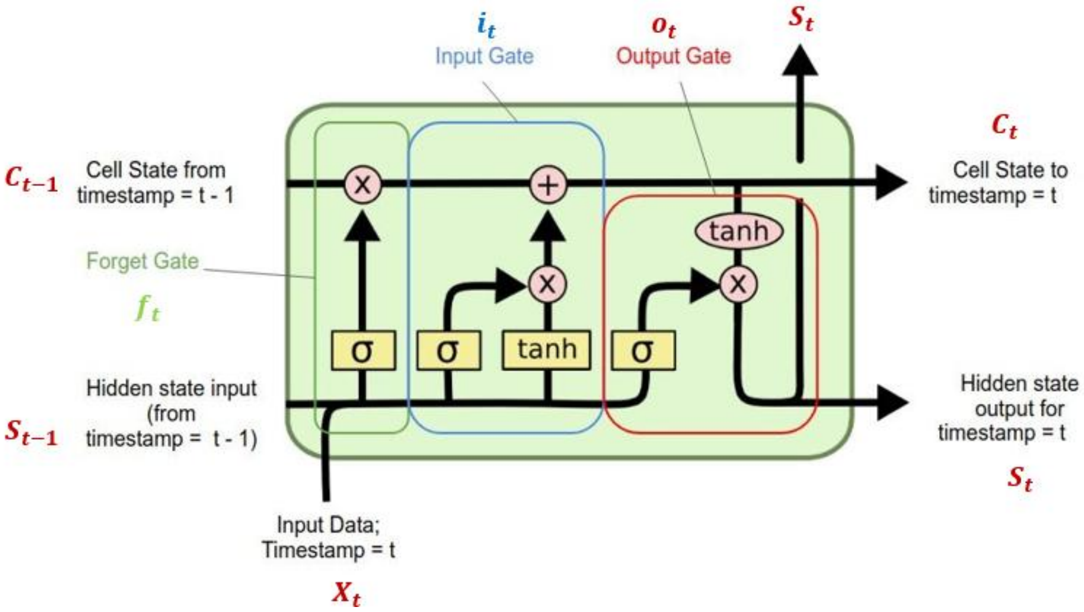
Jon M. Huntsman School of Business
UtahStateUniversity

Pedram Jahangiry

# Let’s try LSTM on the temperature example

```python
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.LSTM(16)(inputs)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)
```

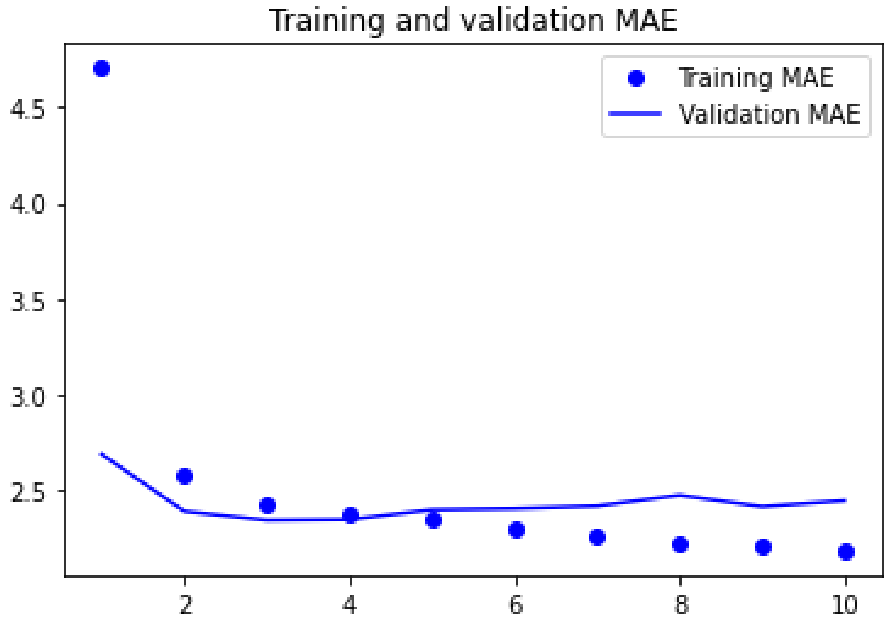
```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_3 (InputLayer)        [(None, 120, 14)]         0

 lstm (LSTM)                 (None, 16)                1984

 dense_3 (Dense)             (None, 1)                 17

=================================================================
Total params: 2,001
Trainable params: 2,001
Non-trainable params: 0
_________________________________________________________________
```




```
output_t = activation(dot(state_t, Uo) + dot(input_t, Wo) + dot(c_t, Vo) + bo)
i_t = activation(dot(state_t, Ui) + dot(input_t, Wi) + bi)
f_t = activation(dot(state_t, Uf) + dot(input_t, Wf) + bf)
k_t = activation(dot(state_t, Uk) + dot(input_t, Wk) + bk)
```

# LSTM performance

- Baseline Test MAE = 2.62
- Simple LSTM Test MAE = 2.53
- Also beats the naïve forecaster.
- Overfitting?



# Can we do better?

# Improving the simple LSTM model


- We can improve the performance of the simple LSTM model by:

1. Recurrent Dropout : use drop out to fight overfitting in the recurrent layers (in addition to drop out for the dense layers)
2. Stacking recurrent layers: increase model complexity to boost representation power
3. Using bidirectional RNN: processing the same information differently! Mostly used in NLP.

Jon M. Huntsman School of Business
UtahStateUniversity

Pedram Jahangiry

# Regular vs Recurrent Dropout

**Regular Dropout vs. Recurrent Dropout**

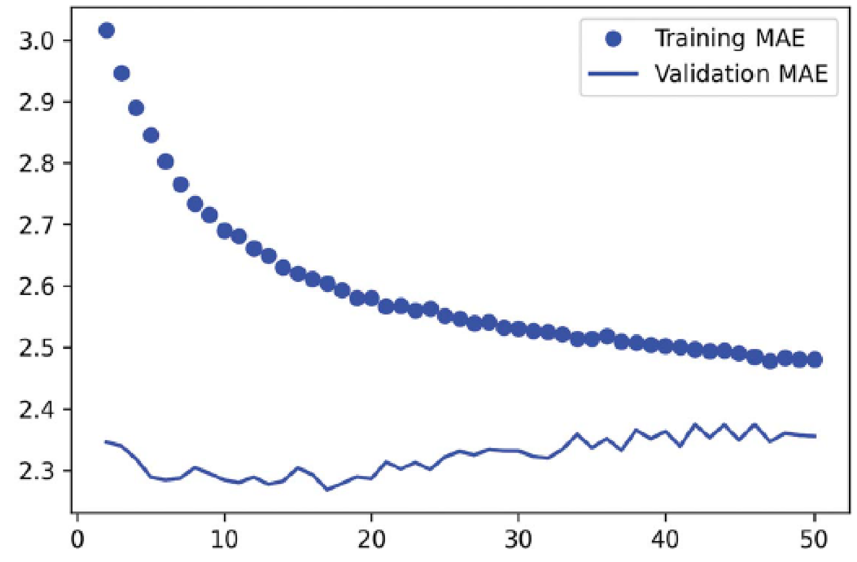
- **Regular Dropout:**
  - Applied to inputs/outputs of RNN layers
  - Example: Randomly dropping elements from an input vector $[x_1, x_2, x_3]$
- **Recurrent Dropout:**
  - Applied to the hidden state (the connection between time steps)
  - Example: Dropping parts of $h_{t-1}$ before computing $h_t$ (e.g., $[h_1, h_2, h_3]$ becomes $[h_1, 0, h_3]$)
- **Purpose:**
  - Both help prevent overfitting
  - Recurrent dropout specifically regularizes temporal memory

# Recurrent Drop out

- The same dropout pattern should be applied at every timestep

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.LSTM(32, recurrent_dropout=0.25)(inputs)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)
```

- Baseline Test MAE = 2.62
- Simple RNN, Test MAE = 2.51
- Simple LSTM, Test MAE = 2.53
- LSTM with dropout, Test MAE = 2.45

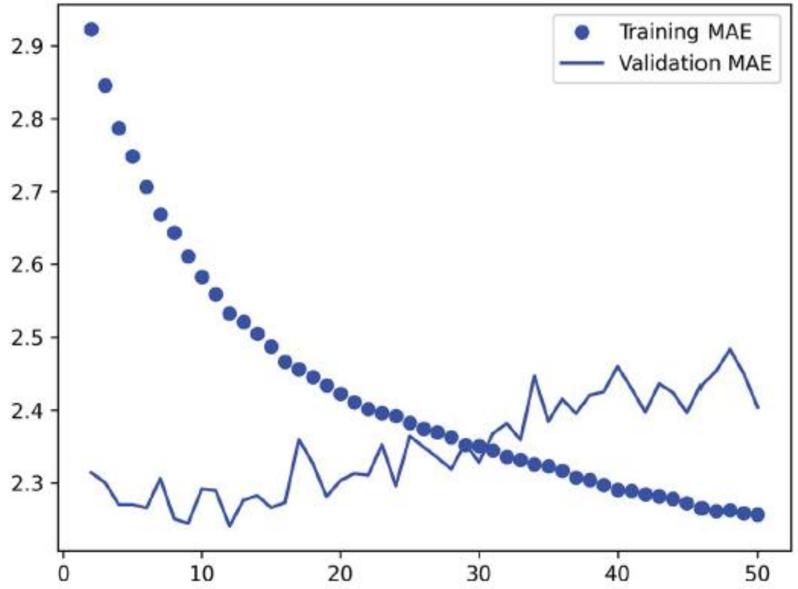


# Stacking Recurrent Layers

- Let’s train a dropout-regulated, stacked GRU model.
- GRU is a slightly simpler version (hence, faster) of LSTM architecture

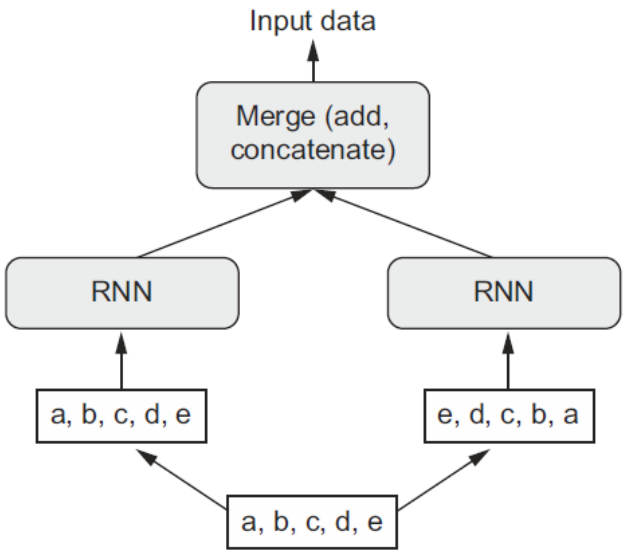
```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.GRU(32, recurrent_dropout=0.5, return_sequences=True)(inputs)
x = layers.GRU(32, recurrent_dropout=0.5)(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)
```

- Baseline Test MAE = 2.62
- Simple RNN, Test MAE = 2.51
- Simple LSTM, Test MAE = 2.53
- Stacking GRU, Test MAE = 2.39



# Bidirectional RNN

- Bidirectional RNN process the input sequence both chronologically and antichronologically.
- Idea: capturing patterns (representations) that might be overlooked by a unidirectional RNN.




- For the temperature example, the bidirectional LSTM strongly underperforms even the common-sense baseline.

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Bidirectional(layers.LSTM(16))(inputs)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)
```

- Baseline Test MAE = 2.62
- Simple RNN, Test MAE = 2.51
- Simple LSTM, Test MAE = 2.53
- Stacking GRU, Test MAE = 2.39
- Bidirectional RNN, Test MAE= 2.79

# Final message

- Deep learning is more an art than science! Too many moving part!
  - Number of units in each recurrent layer
  - Number of stacked layers
  - Amount of dropout and recurrent dropout
  - Number of dense layers
  - Sequence horizon!
  - Optimizers, learning rates and etc
  - ….


- Apply RNN to datasets that past is a good predictor of the future! Not the stock market!

# Road map!

- ✓ Module 1- Introduction to Deep Forecasting
- ✓ Module 2- Setting up Deep Forecasting Environment
- ✓ Module 3- Exponential Smoothing
- ✓ Module 4- ARIMA models
- ✓ Module 5- Machine Learning for Time series Forecasting
- ✓ Module 6- Deep Neural Networks
- ✓ Module 7- Deep Sequence Modeling (RNN, LSTM)
- Module 8- Prophet and Neural Prophet

Jon M. Huntsman School of Business
Utah State University

Pedram Jahangiry