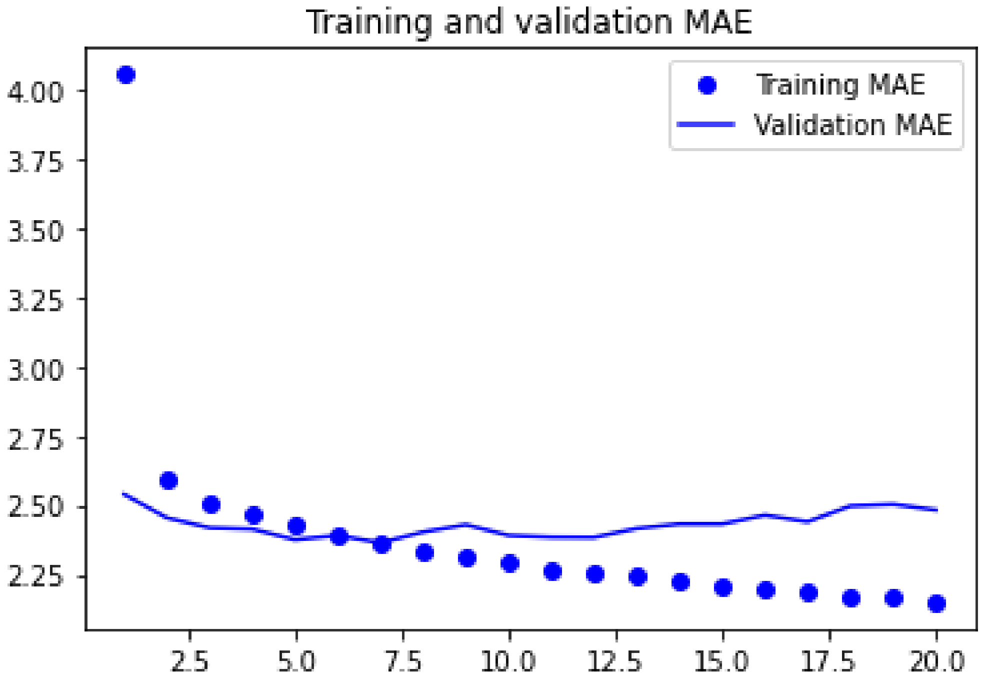
Recurrent Neural Networks

CSE 4311 – Neural Networks and Deep Learning

Vassilis Athitsos
Computer Science and Engineering Department
University of Texas at Arlington

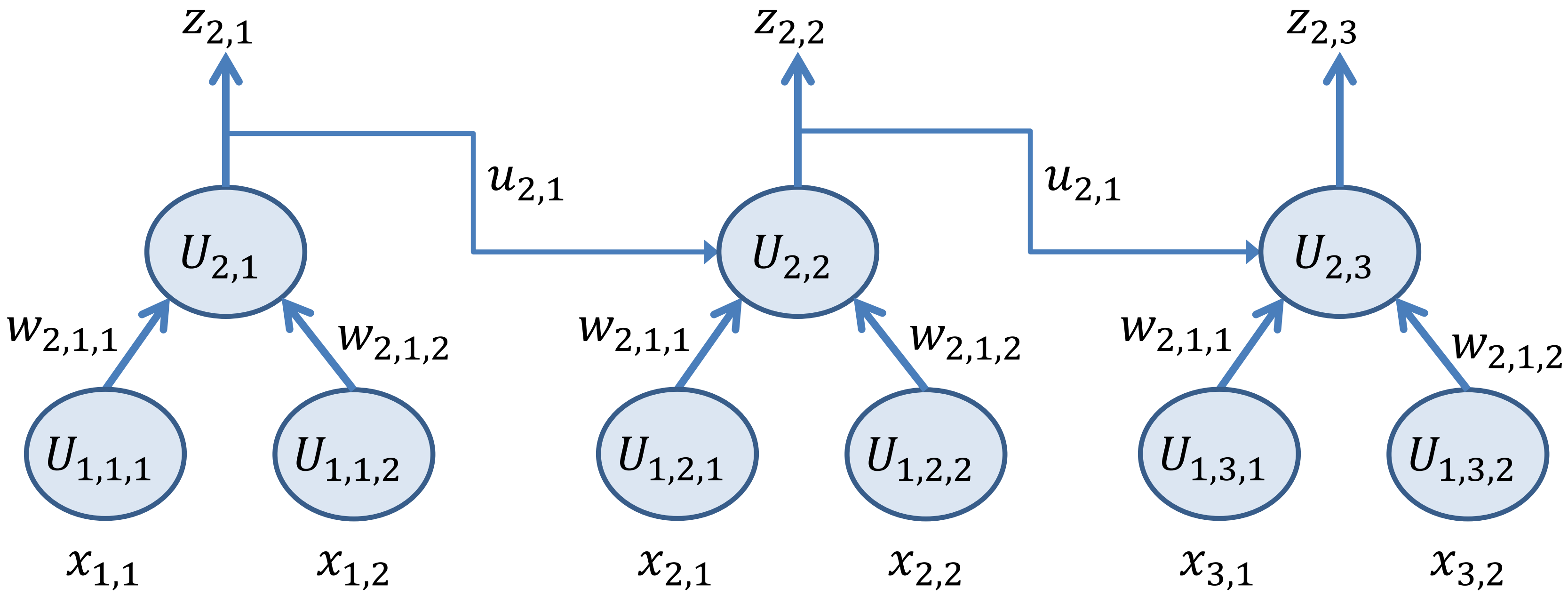
Jena Climate: RNN Results



- Validation MAE:
  - RNN: 2.31 to 2.34.
  - Simple fully-connected network: ranged from 2.54 to 2.66.
  - “Naïve” method: 2.46.
- Test MAE:
  - RNN: 2.45-2.48.
  - Simple fully-connected network: ranged from 2.63 to 2.71.
  - “Naïve” method: 2.62.
- The improvement is not really spectacular.
  - In other datasets, bigger (or smaller) differences in accuracy can be observed.

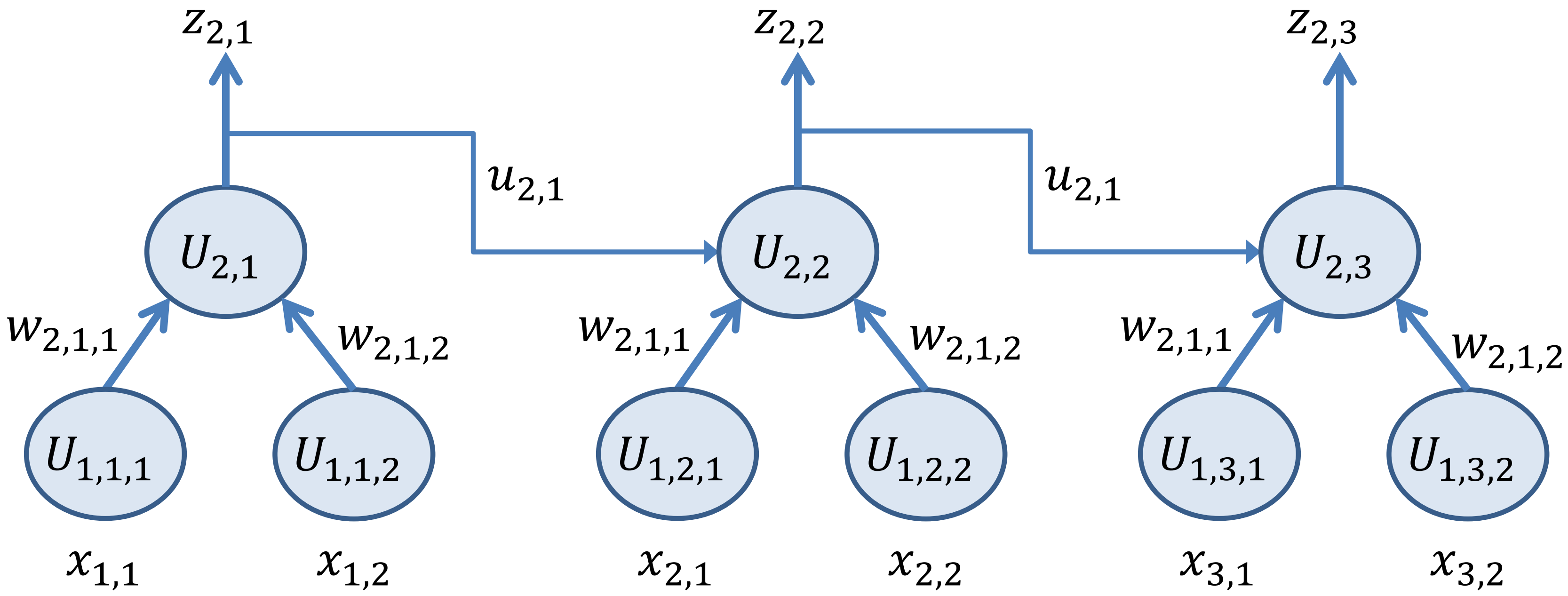
A Simple RNN Model

- This is an example model that is small enough to draw easily:
  - The input to the model is a time series $\boldsymbol{x}$ of length 3.
  - Each element of the $\boldsymbol{x}$ has two dimensions.
- So, $\boldsymbol{x} = \big((x_{1,1}, x_{1,2}), (x_{2,1}, x_{2,2}), (x_{3,1}, x_{3,2})\big)$



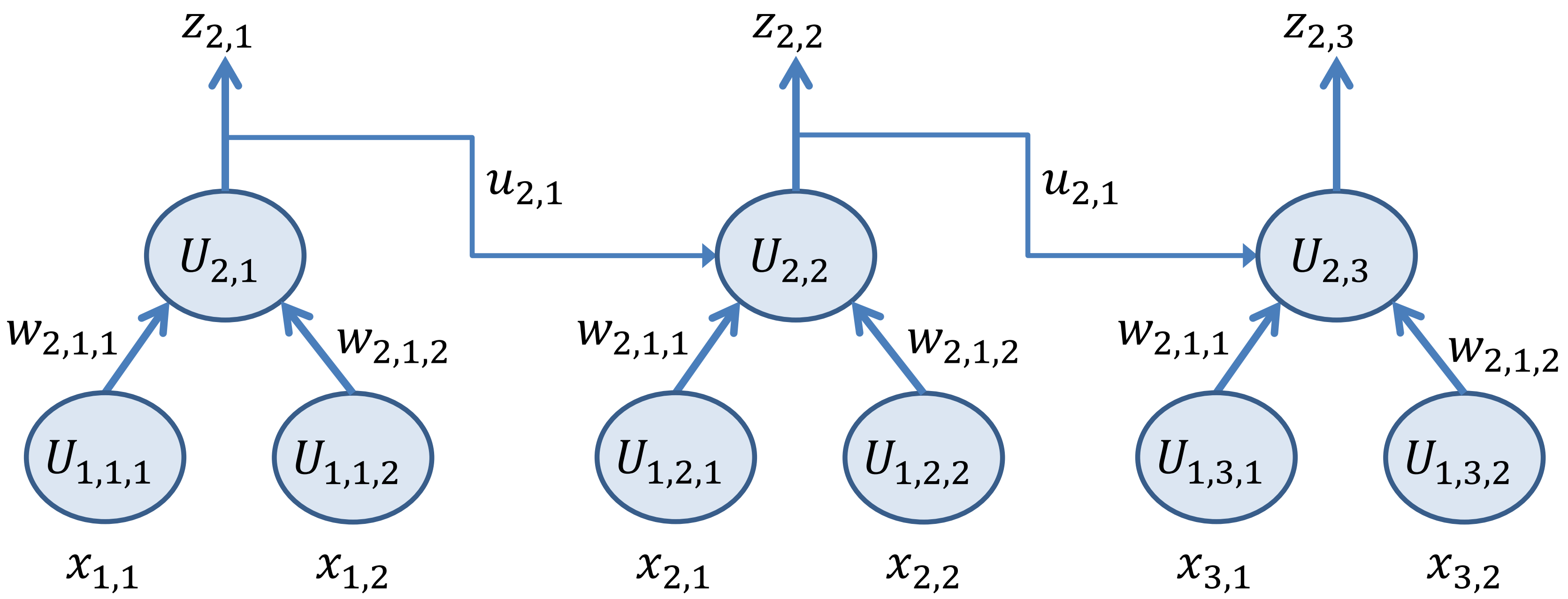
A Simple RNN Model

- Previously, we used to draw input layers on the left, and output layers on the right.
- Here, it is easier to draw input layers at the bottom, and output layers at the top.



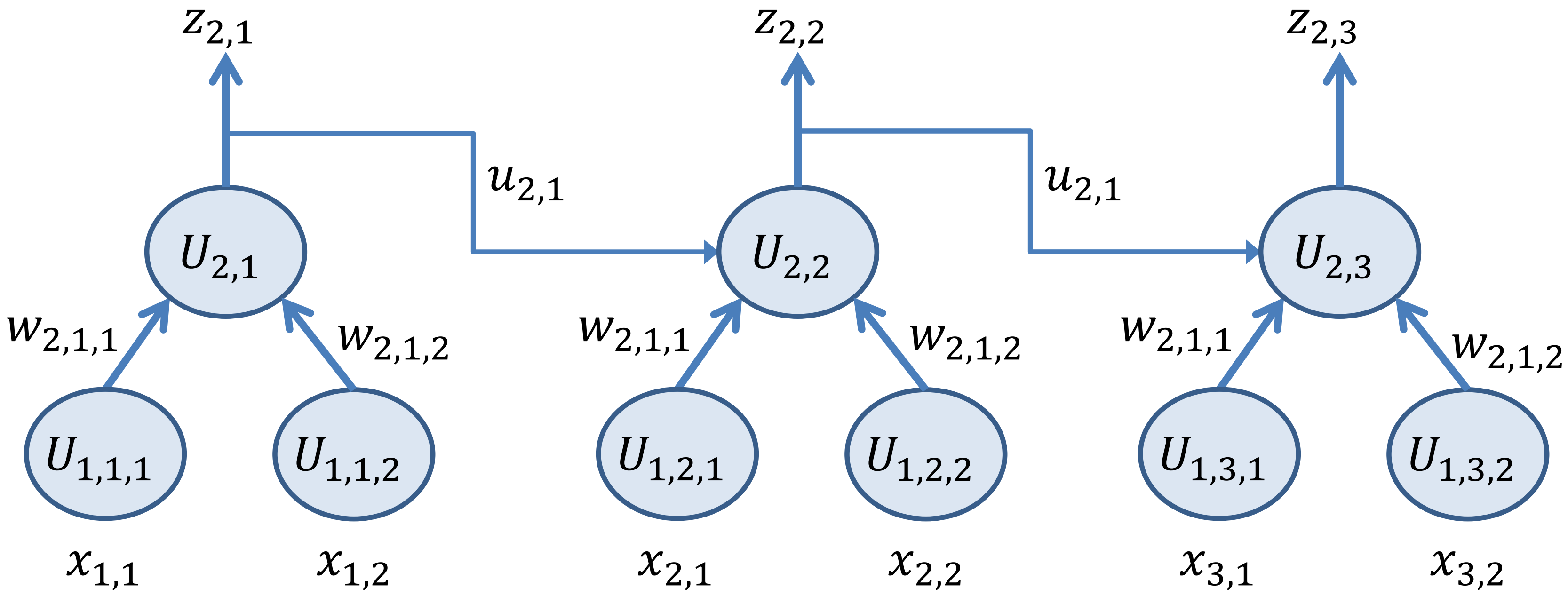
A Simple RNN Model

- There is nothing different about the input layer, it is as usual.
- $\boldsymbol{x} = \left(\left(x_{1,1}, x_{1,2}\right), \left(x_{2,1}, x_{2,2}\right), \left(x_{3,1}, x_{3,2}\right)\right)$, so we need six input units to represent the input.



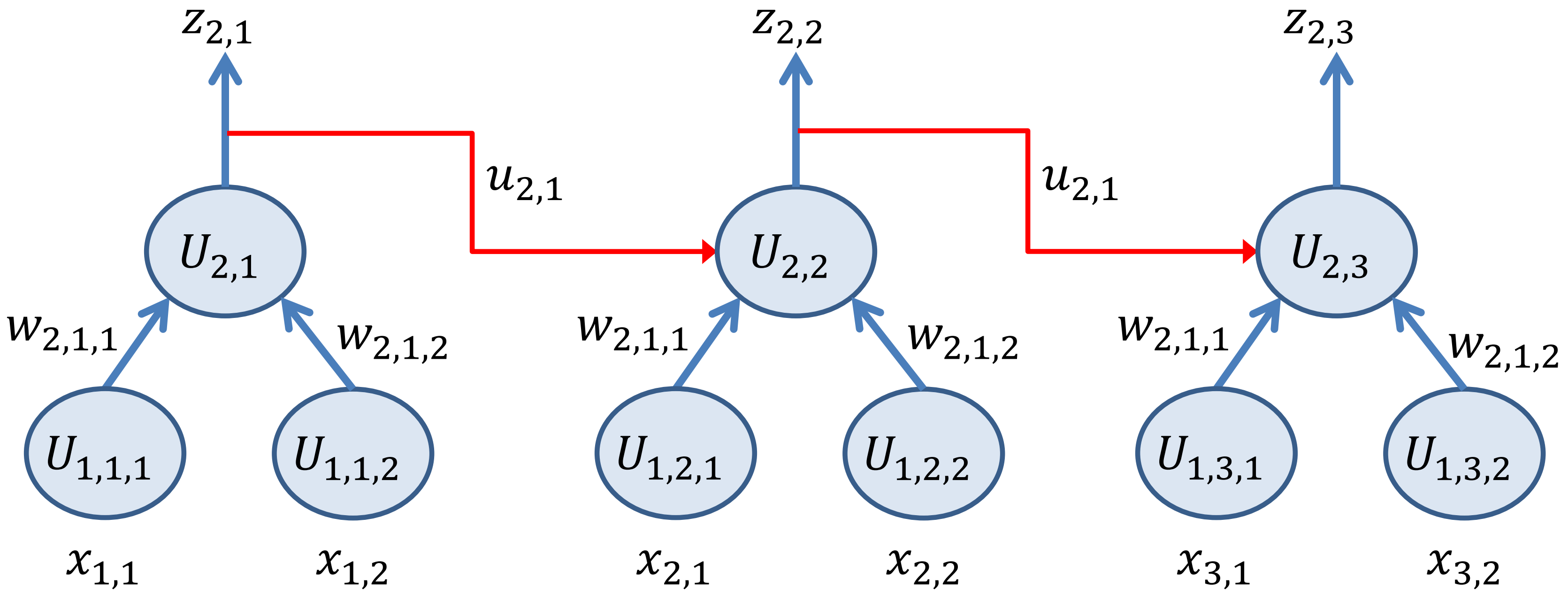
A Simple RNN Model

- The second layer is a **recurrent** layer.
  - Because of this layer, **this network is not a feedforward neural network**.
  - In a feedforward neural network, the inputs to a layer come from the outputs of the previous layer.
  - Here, some inputs to the second layer come from the second layer itself.



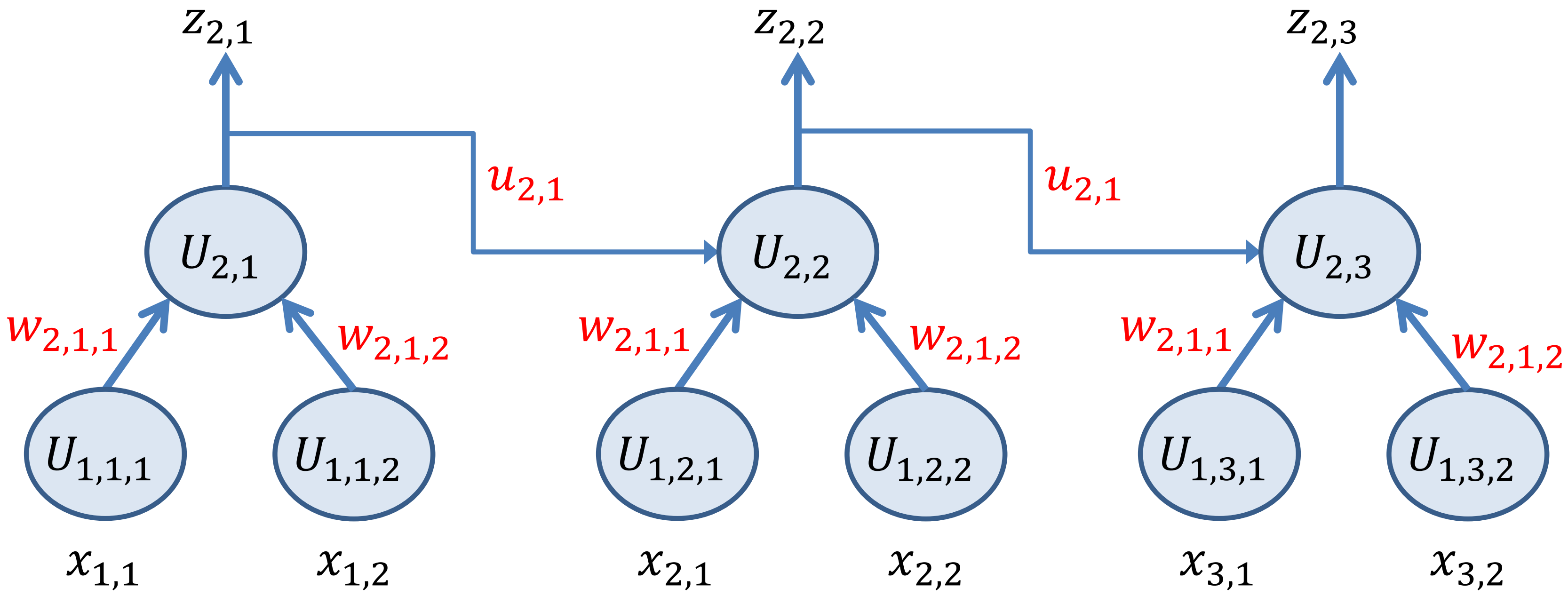
A Simple RNN Model

- Notice that unit $U_{2,2}$ receives inputs not only from input units, but also from second-layer unit $U_{2,1}$.
- Similarly, unit $U_{2,3}$ receives inputs not only from input units, but also from second-layer unit $U_{2,2}$.
- These connections between units of the same layer are called **recurrent connections.**



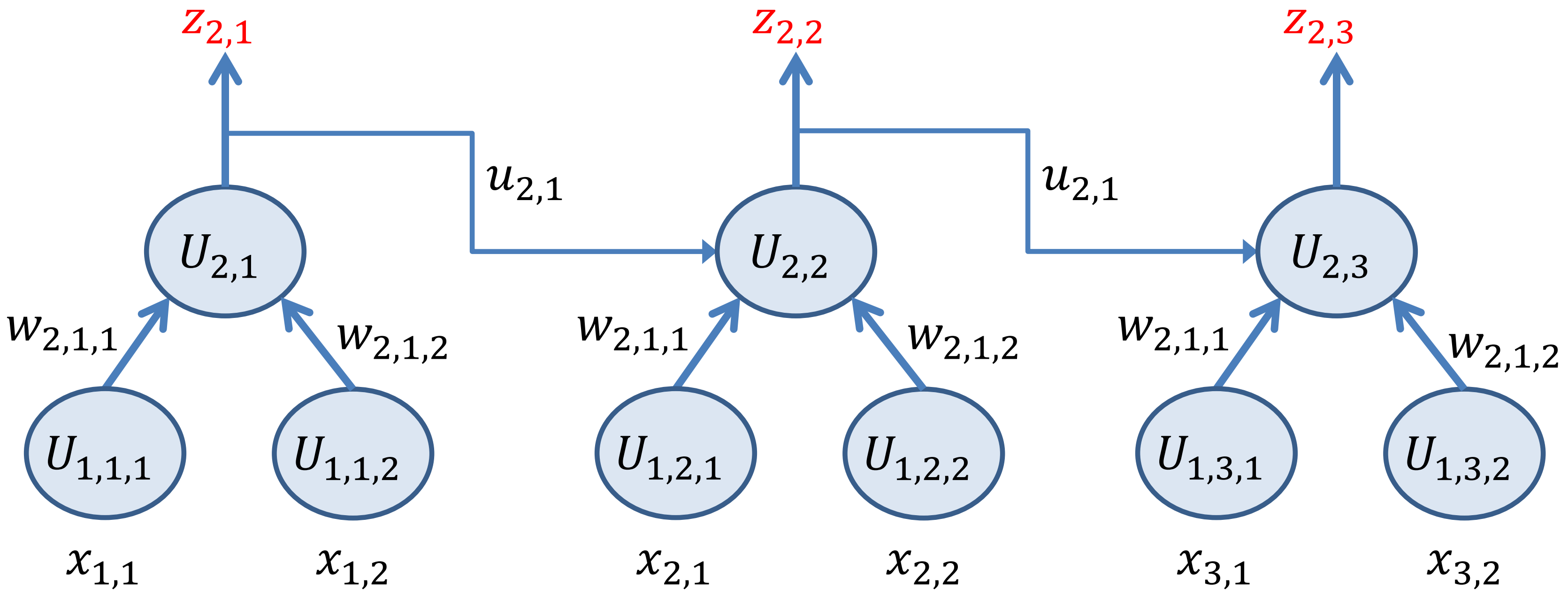
A Simple RNN Model

- Notice that all units in the second layer share the same weights.
- The weights connecting two input units to a second-layer unit are denoted with the same two symbols.
- There is also a new symbol, the **recurrent weight** $u_{2,1}$ for the recurrent connections between $U_{2,1}$ and $U_{2,2}$, and between $U_{2,2}$ and $U_{2,3}$.



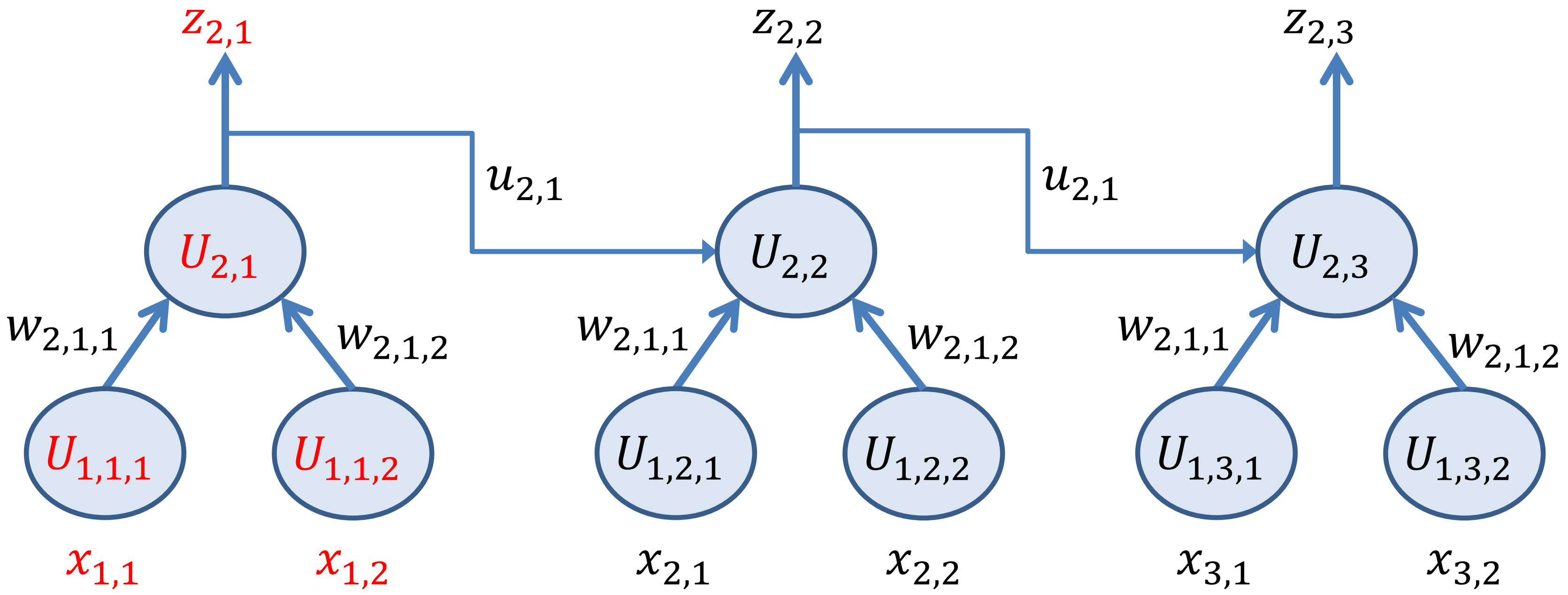
Computing the Output

- The outputs of the second layer play two roles:
  - They are used as inputs to other units in the second layer.
  - They are also the outputs of the entire network.
- In more complicated models, we could have more layers on top.



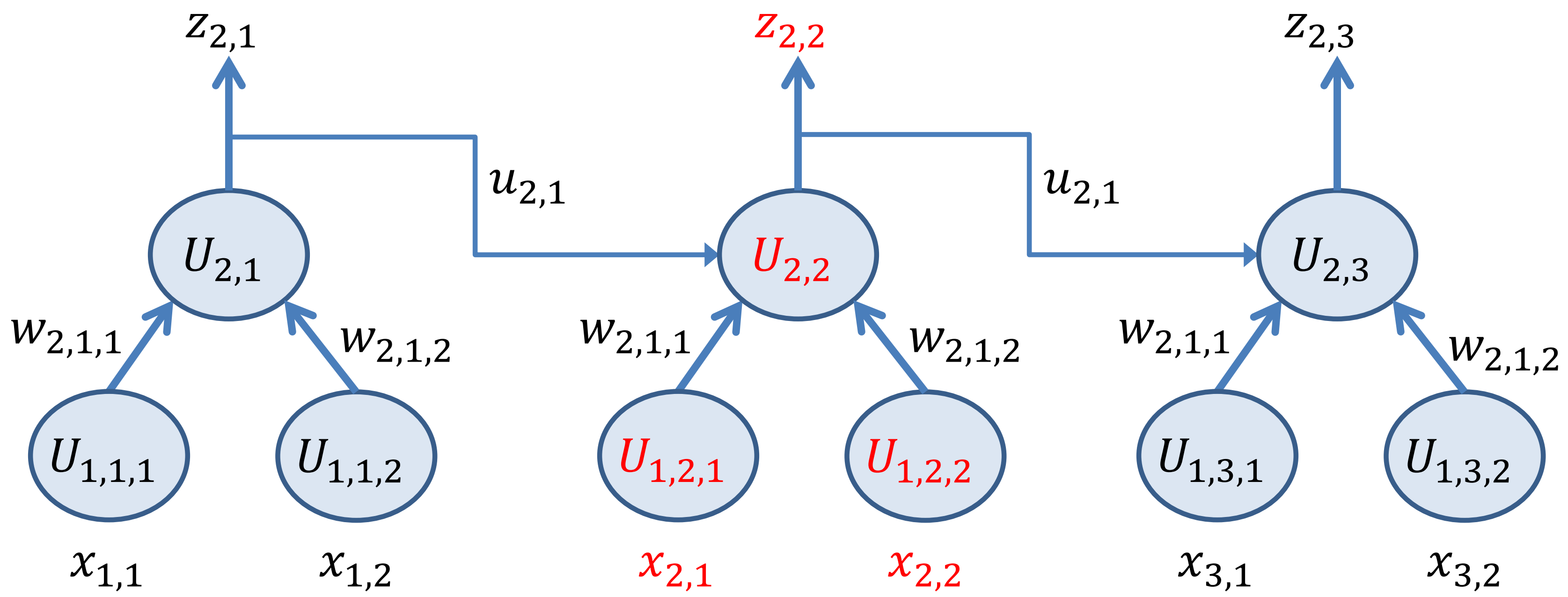
Computing the Output

- Computing the output of each unit needs to be follow the order of the time steps.
- First we compute, from bottom to top, the output of all units that correspond to time step 1.



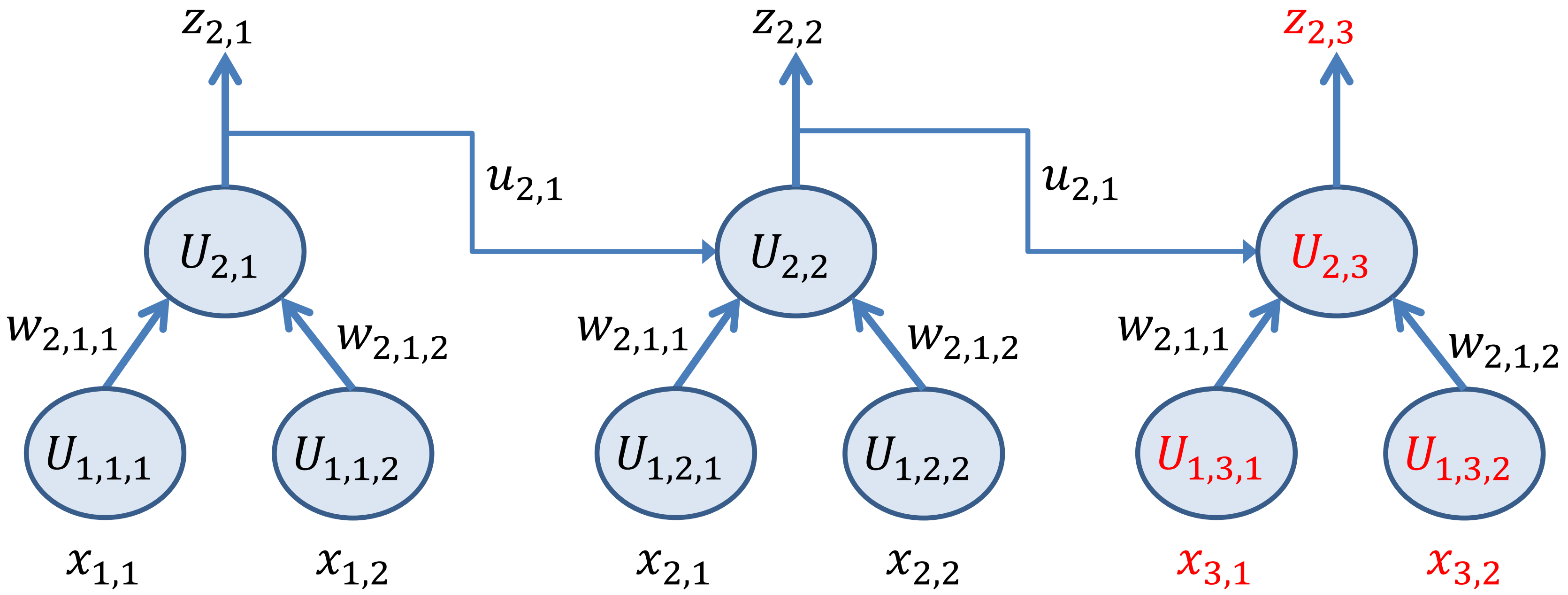
Computing the Output

- Computing the output of each unit needs to be follow the order of the time steps.
- Second we compute, from bottom to top, the output of all units that correspond to time step 2.
  - This way we can use $z_{2,1}$ from time step 1.



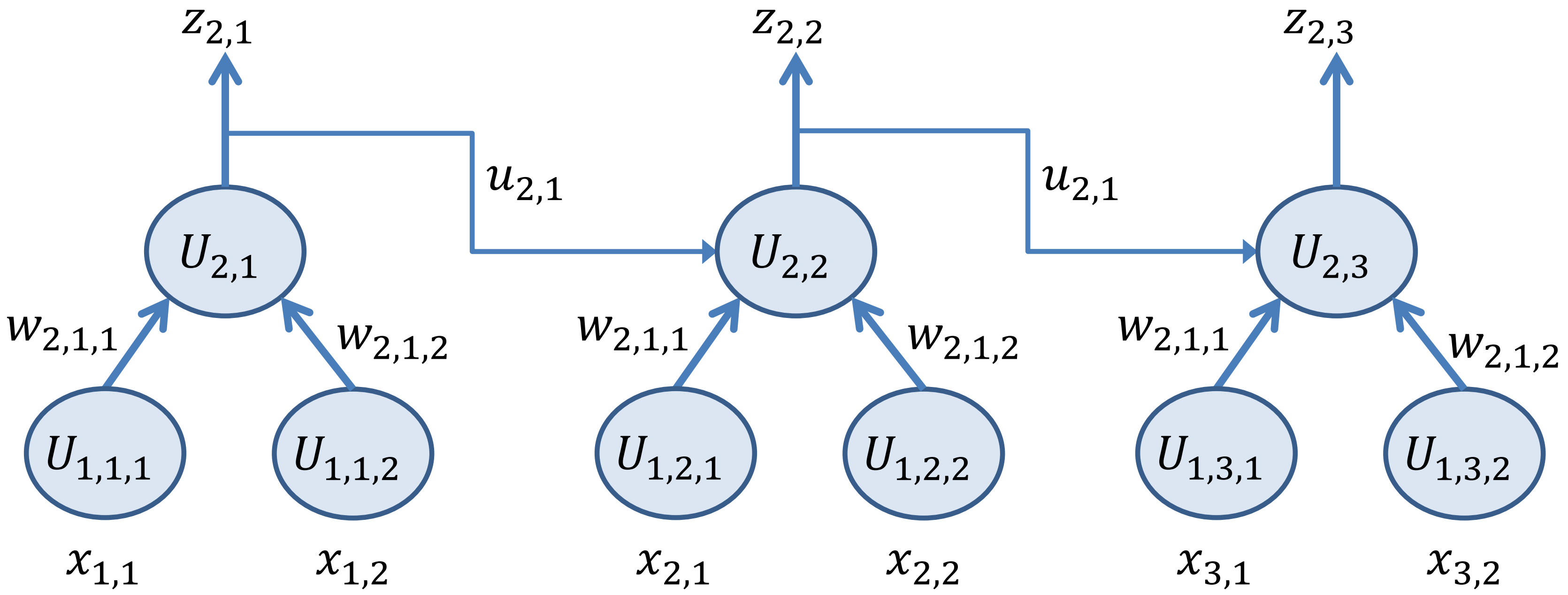
Computing the Output

- Computing the output of each unit needs to be follow the order of the time steps.
- Third we compute, from bottom to top, the output of all units that correspond to time step 3.
  - This way we can use $z_{2,2}$ from time step 2.



Order of Output Computations

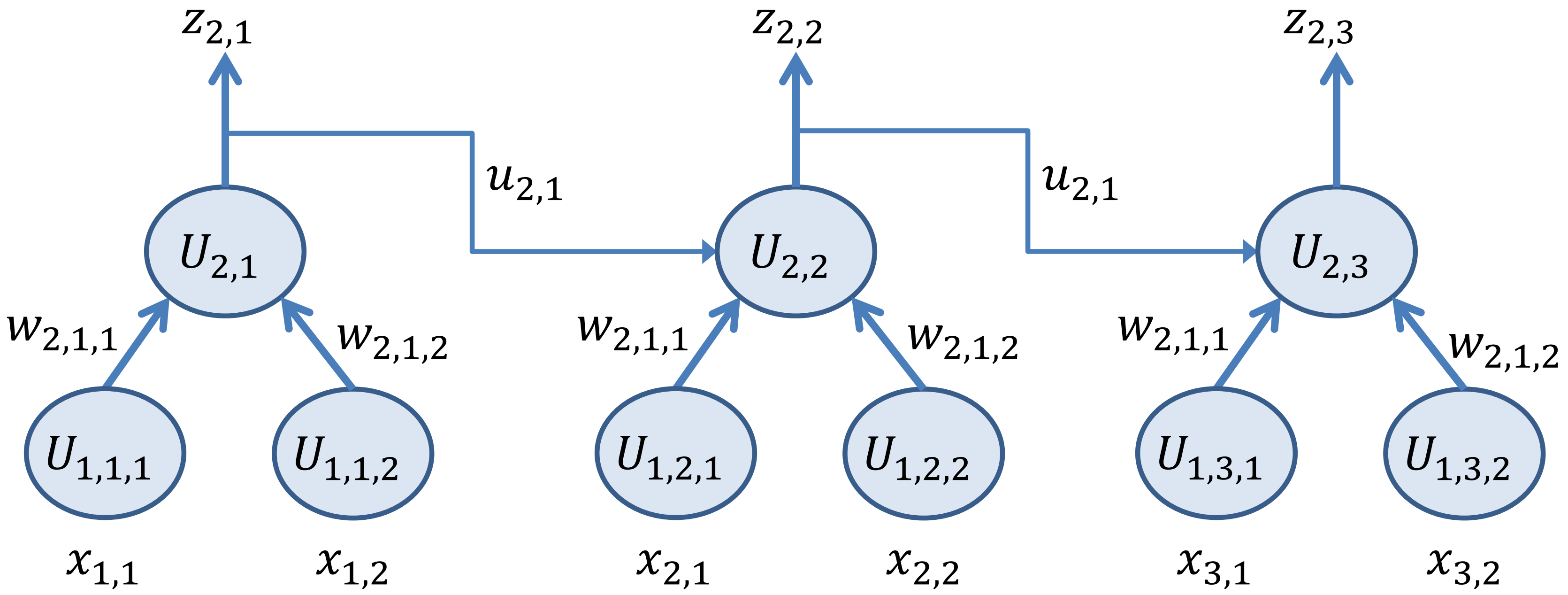
- In a feedforward neural network, we simply followed the order of layers, from input to output.
- In an RNN, we first follow the order of time steps.
  - Within a single time step, we follow the order of layers, from input to output.



Translating to Keras

```
model = keras.Sequential([keras.Input(shape=(3,2)),
                          keras.layers.SimpleRNN(1)])
```

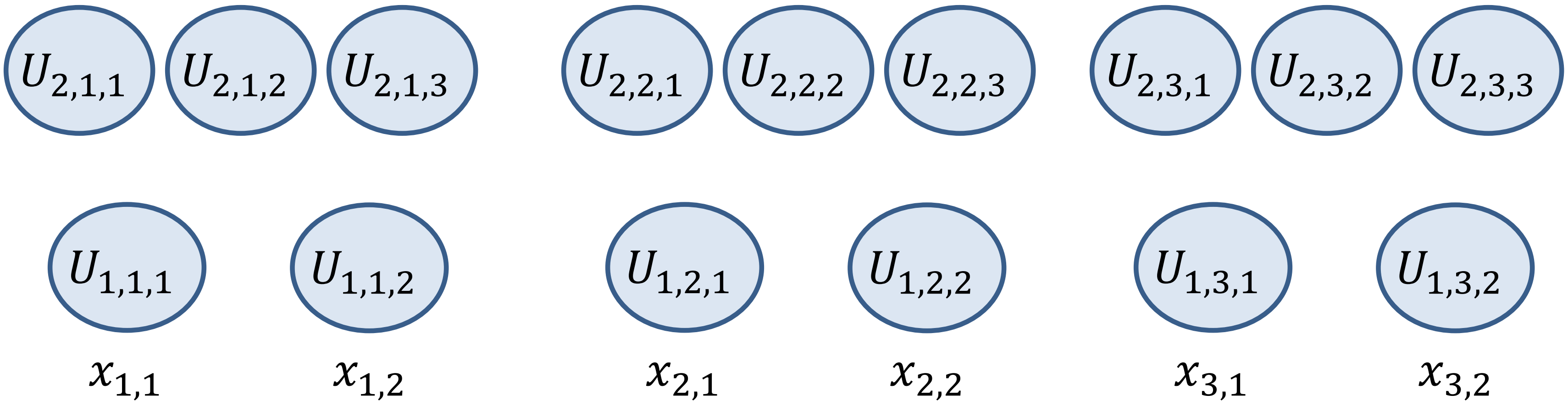
- This piece of code implements our network.
  - Parameter 1 for the SimpleRNN layer specifies one unit per time step.



Translating to Keras

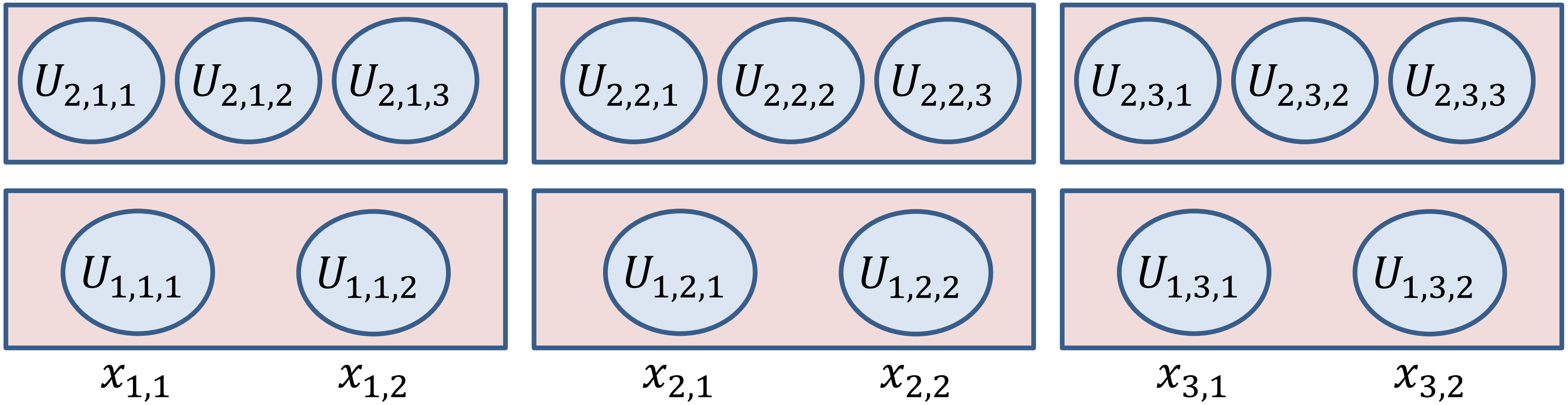
```
model = keras.Sequential([keras.Input(shape=(3,2)),
                          keras.layers.SimpleRNN(3)])
```

- Here the SimpleRNN layer has three units per time step.
- We do not show the connections and weights anymore.
  - Within a time step, all three 2nd layer units are connected to all two inputs.
  - All 2nd layer units from the previous step are inputs to all 2nd layer units in the next step.



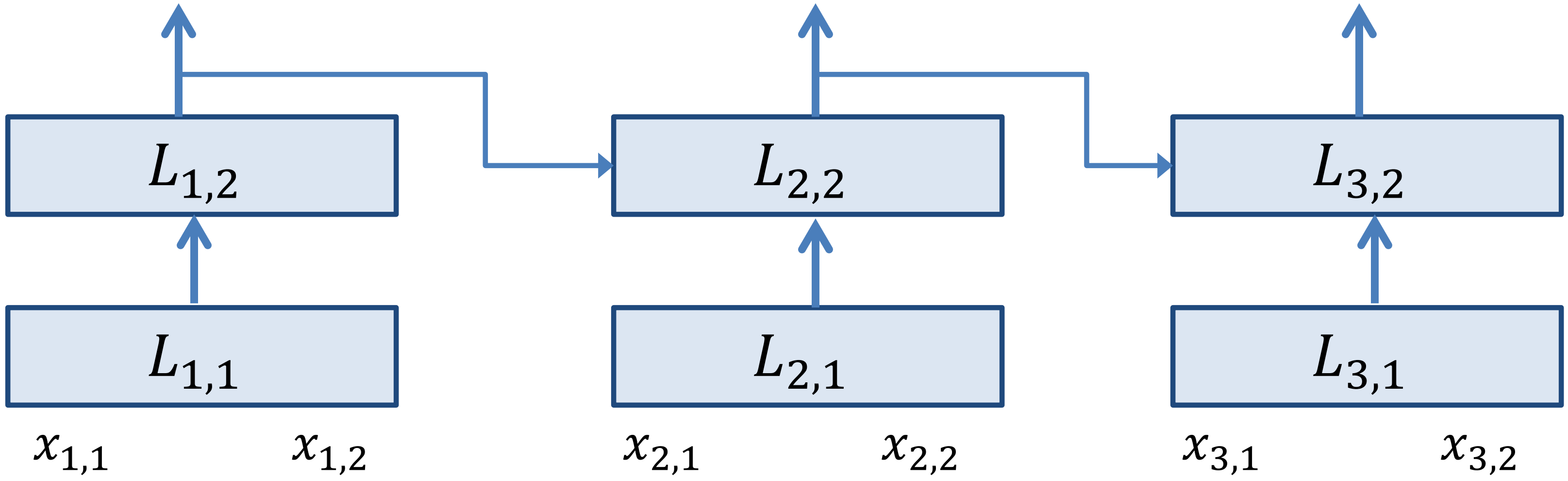
Simplifying the Drawings

- When we draw an RNN, we typically do not show each individual unit.
- Instead, we group units into blocks, such that all units in a block belong to the same layer and the same time step.



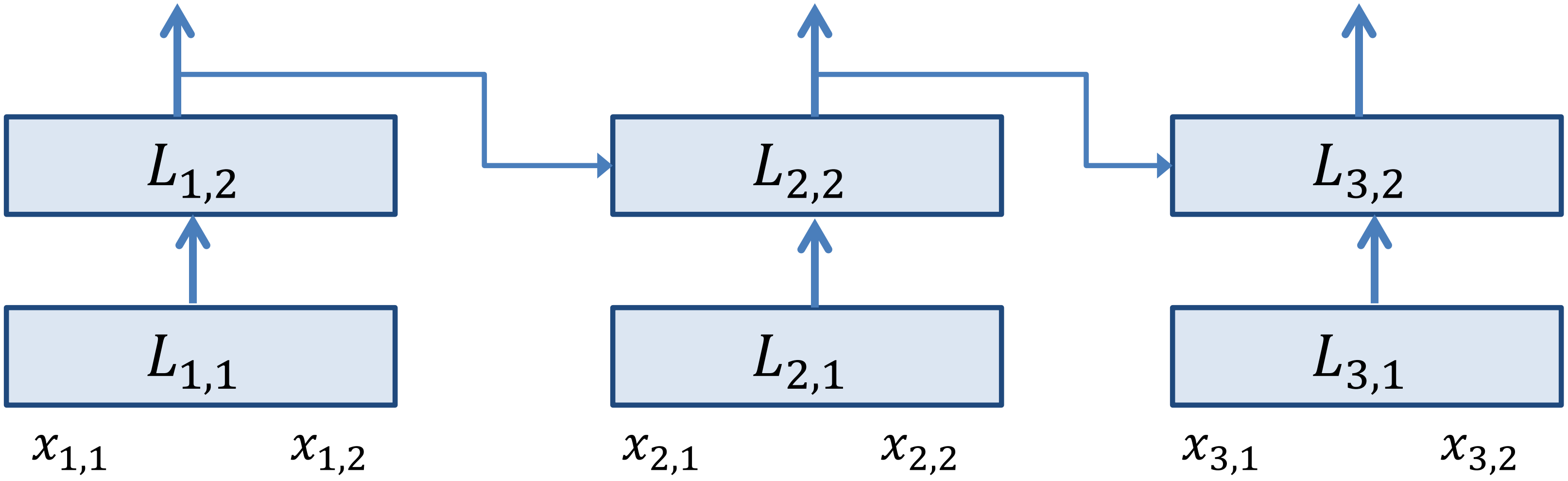
Simplifying the Drawings

- When we draw an RNN, we typically do not show each individual unit.
- Instead, we group units into blocks, such that all units in a block belong to the same layer and the same time step.
- In this simplified drawing:
  - $L_{1,1}$ groups together units $U_{1,1,1}$ and $U_{1,1,2}$.
  - $L_{1,2}$ groups together units $U_{2,1,1}$, $U_{2,1,2}$, and $U_{2,1,3}$.



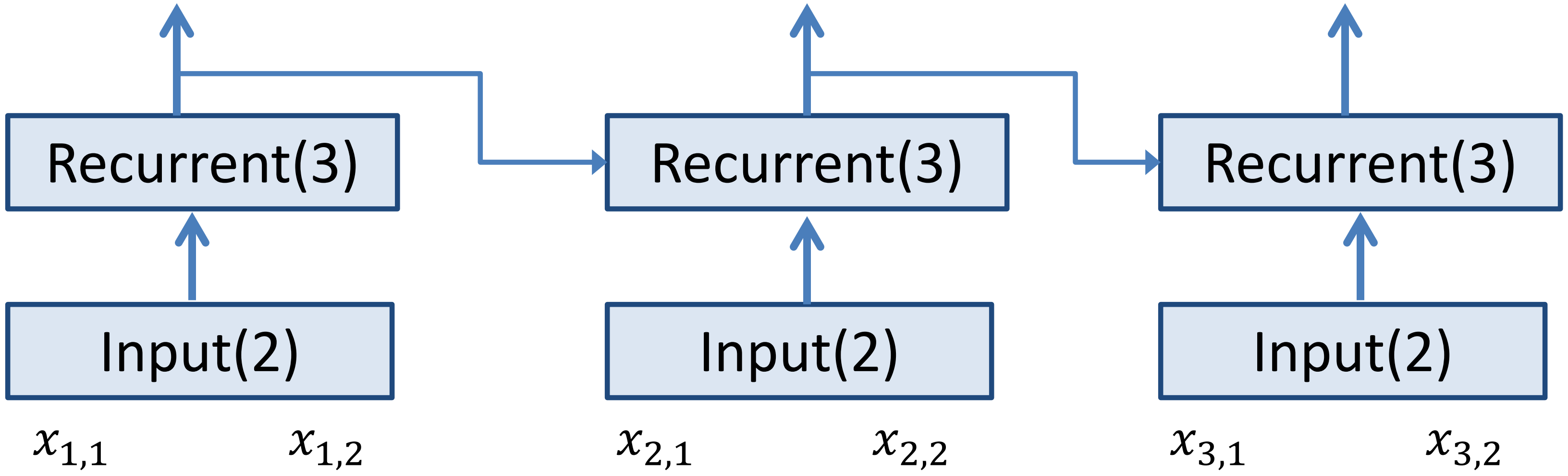
Simplifying the Drawings

- Now that we have simplified the drawing, we can draw connections again.
- An arrow means that all units of one group are connected to all units of the other group.
- Of course, now it is not clear how many units are in each layer.
  - When we simplify, some details are inevitably lost.



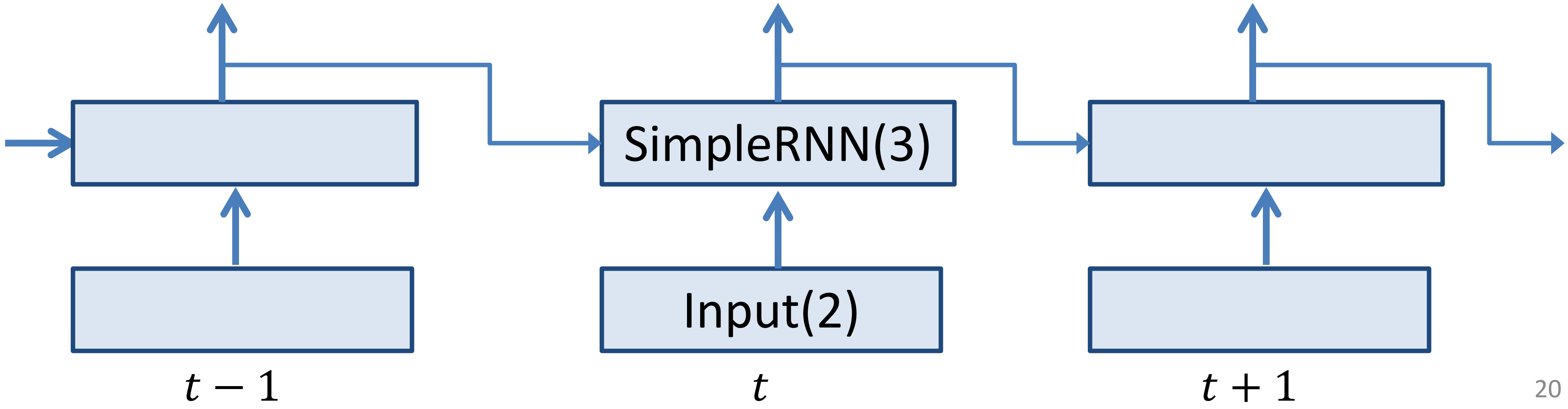
Simplifying the Drawings

- We can always make up conventions to provide more information.
- For example, here we show for each block:
  - The type of layer that it belongs to.
  - The number of units.



Simplifying the Drawings

- This is a common way to draw RNNs.
- Since the structure at each time step looks the same, we just show three steps:
  - “Previous”, “current”, and next”.
  - Oftentimes more detail in shown in the current step.



An RNN Network for Jena Climate

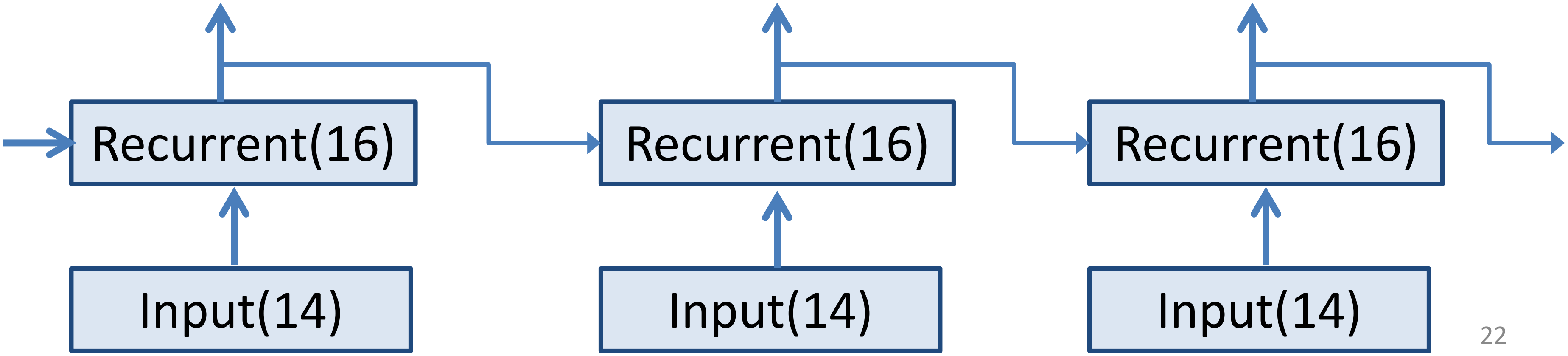
```
model = keras.Sequential([keras.Input(shape=input_shape),
                          keras.layers.SimpleRNN(16),
                          keras.layers.Dense(1),])

model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])

callbacks = [keras.callbacks.ModelCheckpoint("jena_LSTM1_16.keras",
                                             save_best_only=True)]

history_lstm = model.fit(training_inputs, training_targets, epochs=20,
                          validation_data=(validation_inputs, validation_targets),
                          callbacks=callbacks)
```

- This code trains a network with an RNN layer.

An RNN Network for Jena Climate

```
model = keras.Sequential([keras.Input(shape=input_shape),
                          keras.layers.SimpleRNN(16),
                          keras.layers.Dense(1),])
```

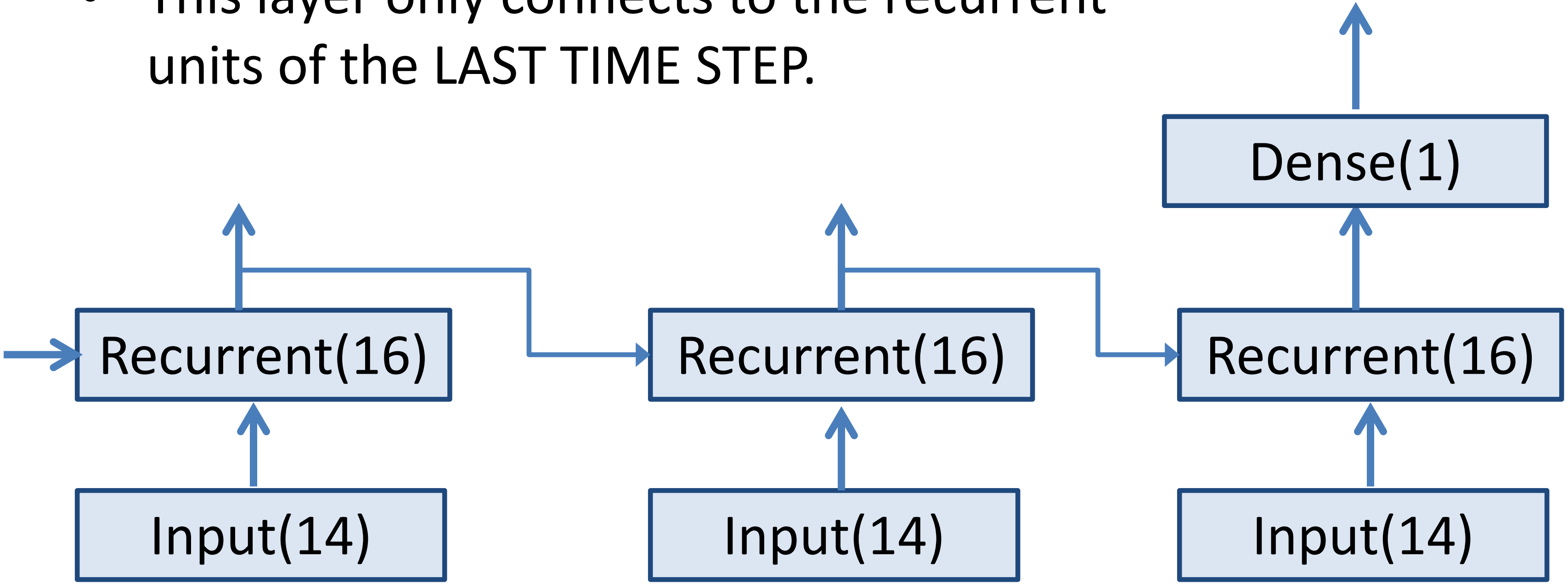
- The highlighted line creates the recurrent layer.
  - It has 16 units for each time step.
  - There are 120 time steps (not shown in this drawing).



An RNN Network for Jena Climate

```
model = keras.Sequential([keras.Input(shape=input_shape),
                          keras.layers.SimpleRNN(16),
                          keras.layers.Dense(1),])
```

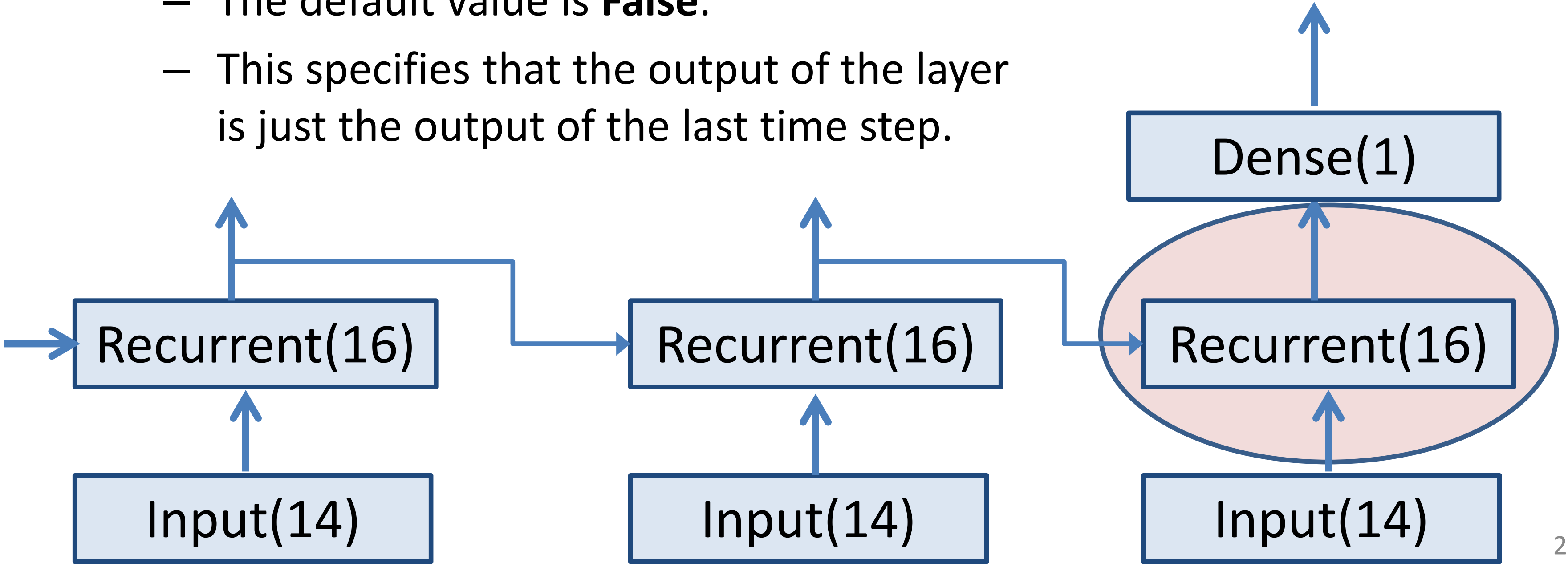
- We have a fully connected output layer.
- This layer only connects to the recurrent units of the LAST TIME STEP.



Detour: The **return_sequences** option

```
model = keras.Sequential([keras.Input(shape=input_shape),
                          keras.layers.SimpleRNN(16, return_sequences=False),
                          keras.layers.Dense(1),])
```

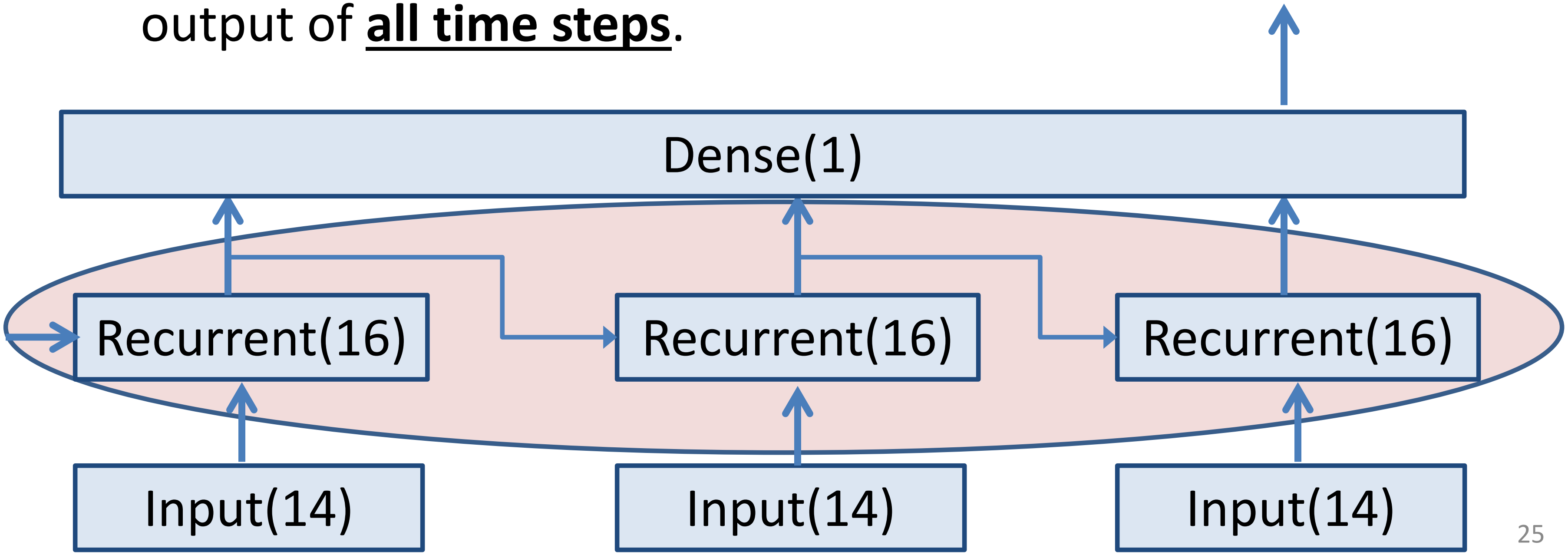
- SimpleRNN layers have an option called **return_sequences**.
  - The default value is **False**.
  - This specifies that the output of the layer is just the output of the last time step.



Detour: The **return_sequences** option

```
model = keras.Sequential([keras.Input(shape=input_shape),
                          keras.layers.SimpleRNN(16, return_sequences=True),
                          keras.layers.Flatten(),
                          keras.layers.Dense(1),])
```

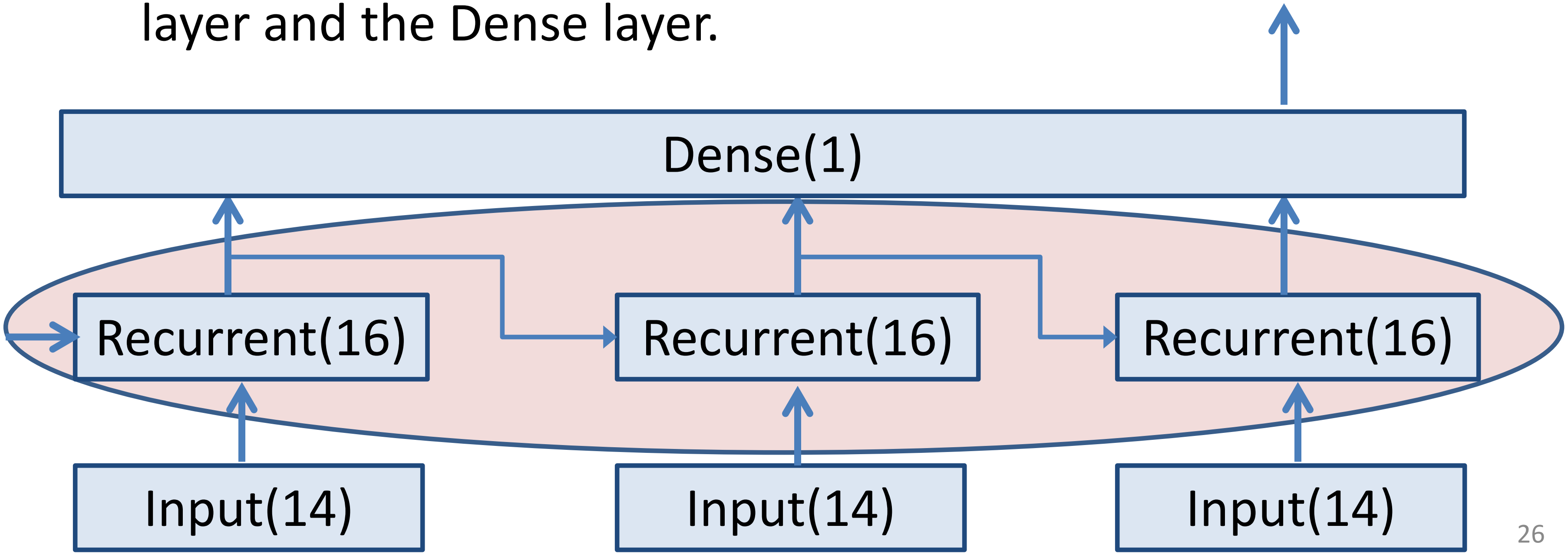
- If **return_sequences** is true, the output of the layer is the output of **all time steps**.



Detour: The **return_sequences** option

```
model = keras.Sequential([keras.Input(shape=input_shape),
                          keras.layers.SimpleRNN(16, return_sequences=True),
                          keras.layers.Flatten(),
                          keras.layers.Dense(1),])
```

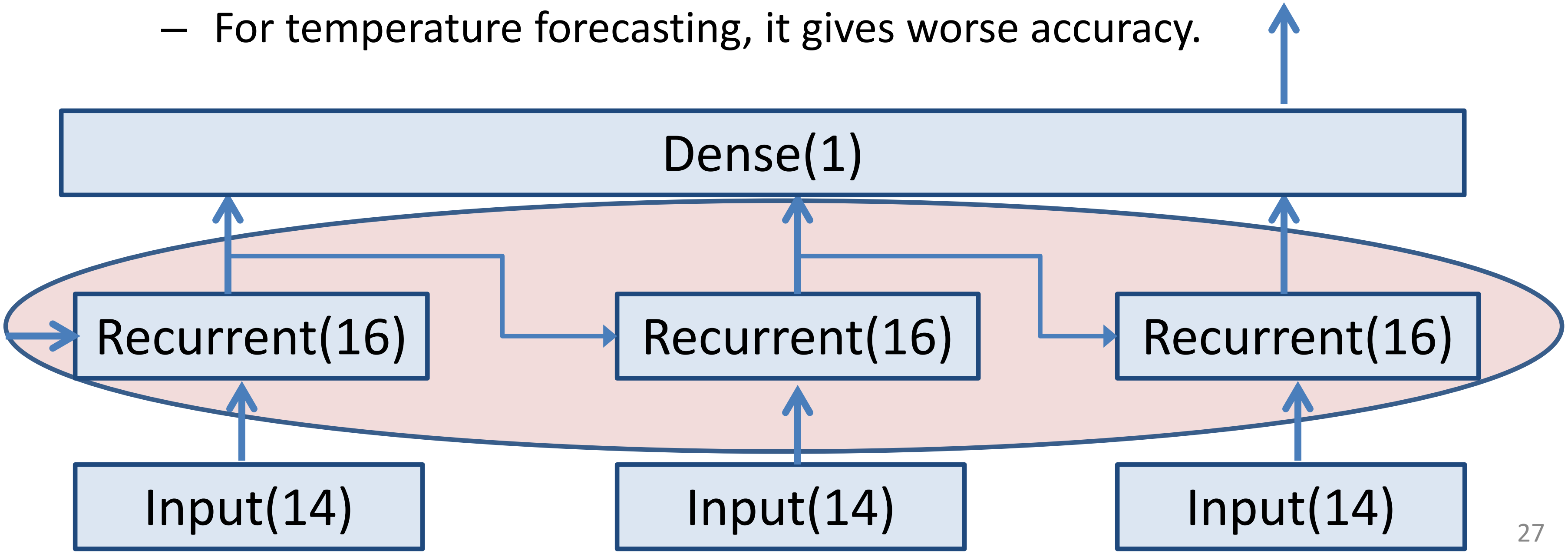
- Note the flattening step in this case, between the SimpleRNN layer and the Dense layer.



Detour: The **return_sequences** option

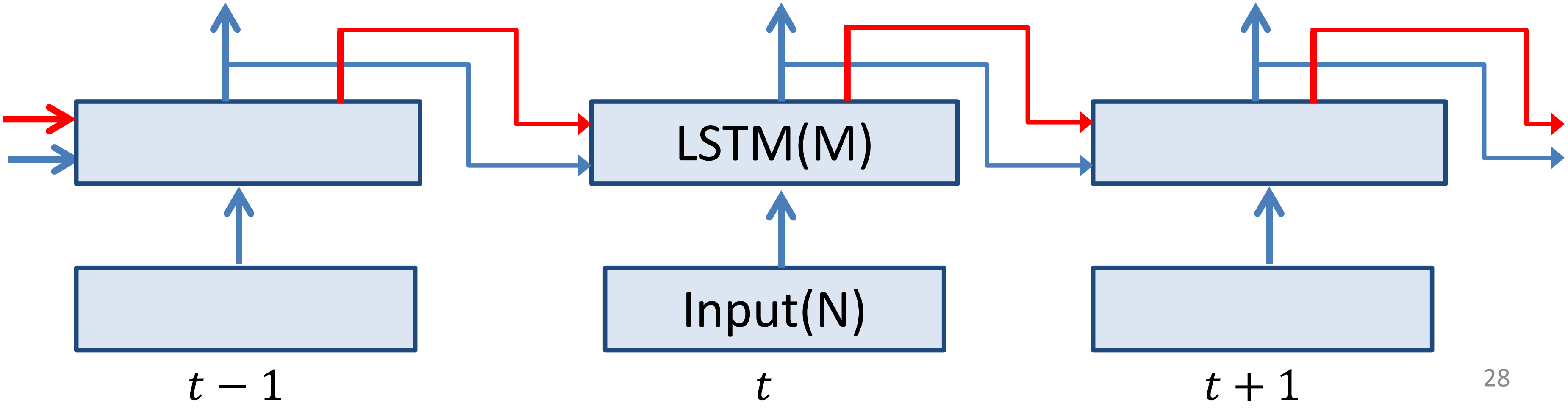
```
model = keras.Sequential([keras.Input(shape=input_shape),
                          keras.layers.SimpleRNN(16, return_sequences=True),
                          keras.layers.Flatten(),
                          keras.layers.Dense(1),])
```

- We will revisit the return_sequences=True option later.
  - For temperature forecasting, it gives worse accuracy.



LSTM Layer

- LSTM stands for Long Short-Term Memory.
- Like SimpleRNN, an LSTM layer is a recurrent layer.
- LSTM layers are used widely in practice.
- However, the description of an LSTM layer is more complicated than that of a SimpleRNN layer.
  - An LSTM layer produces a secondary output, shown in red. This is called the **carry**, and it is computed using some special rules.

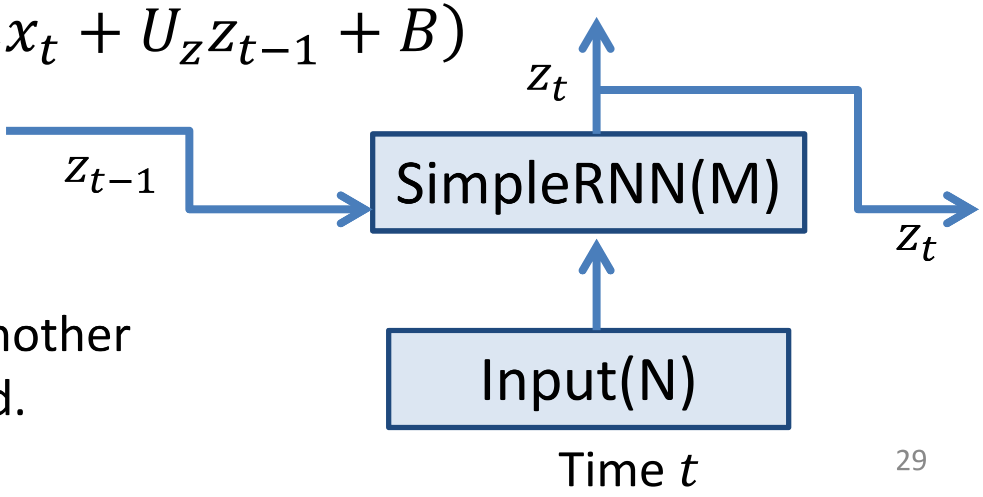


SimpleRNN Computations at Time t

- The output of a simple RNN layer at time t depends on:
  - An N-dimensional input vector x_t from the input layer.
  - An M-dimensional vector z_{t-1}, produced by the RNN layer at time $t-1$.
  - W_z, which is an $M \times N$ matrix of weights applied to x_t.
  - U_z, which is an $M \times M$ matrix of weights applied to z_{t-1}.
  - B_z, which is an M-dimensional vector of bias weights.
- The output z_t is computed as:

$$z_t = \tanh(W_z x_t + U_z z_{t-1} + B)$$

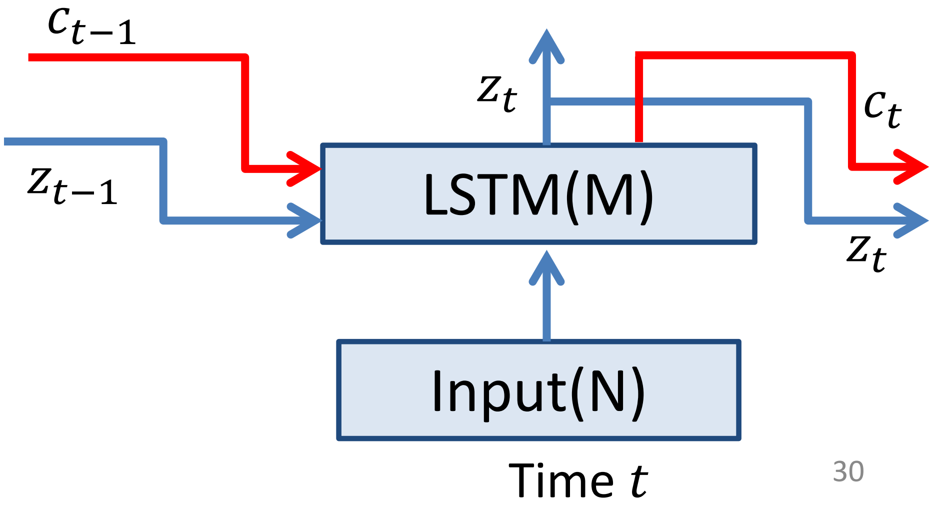
  - Note that tanh is the default activation function for a SimpleRNN layer, but another function could be substituted.



LSTM Computations at Time t

- In addition to the inputs and weights of a simple RNN layer, an LSTM layer also has:
  - An M-dimensional **carry** vector c_{t-1}, produced at time $t-1$.
  - V_z, which is an $M \times M$ matrix of weights applied to c_{t-1}.
- The output z_t is now computed with a new formula:

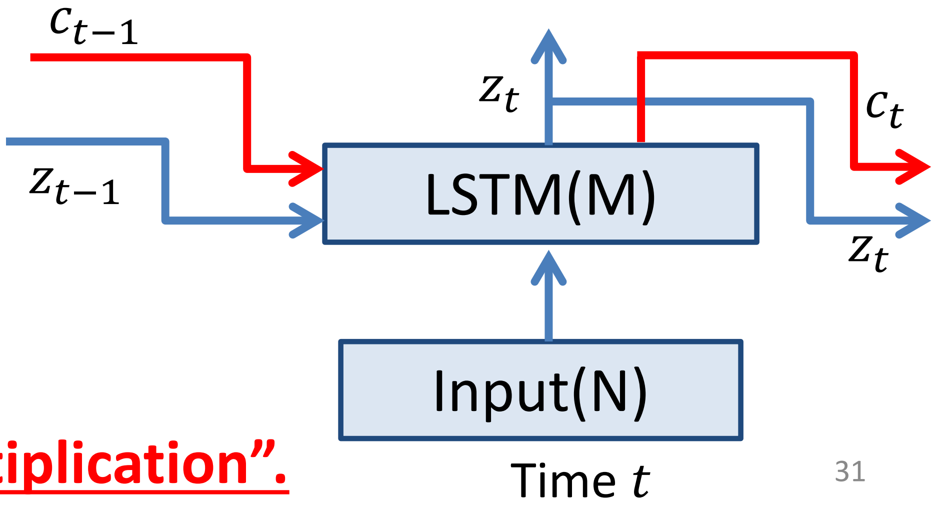
$$z_t = \tanh(W_z x_t + U_z z_{t-1} + V_z c_{t-1} + B)$$



LSTM Computations at Time t

- To complete the description of the LSTM layer, we must specify how to compute the **carry** vector c_t at time t.
- To do that, we use some additional weight matrices:
  - W_i, W_f, W_k are three $M \times N$ weight matrices applied to x_t.
  - U_i, U_f, U_k are three $M \times M$ weight matrices applied to z_t.
  - B_i, B_f, B_k are three M-dimensional vectors of bias weights.
- Then, we compute:

$$i_t = \sigma(W_i x_t + U_i z_{t-1} + B_i)$$
$$k_t = \sigma(W_k x_t + U_k z_{t-1} + B_k)$$
$$f_t = \sigma\left(W_f x_t + U_f z_{t-1} + B_f\right)$$
$$c_t = i_t * k_t + c_{t-1} * f_t$$



Symbol * means "pointwise multiplication".

An Intuitive Interpetation

$$i_t = \sigma(W_i x_t + U_i z_{t-1} + B_i)$$
$$k_t = \sigma(W_k x_t + U_k z_{t-1} + B_k)$$
$$f_t = \sigma\left(W_f x_t + U_f z_{t-1} + B_f\right)$$
$$c_t = i_t * k_t + c_{t-1} * f_t$$

- There is a somewhat intuitive interpretation that motivated these formulas. According to this interpretation:
  - i_t represents new information, computed at time t.
  - k_t represents the importance of each dimension in i_t.
  - c_{t-1} represents old information, computed from previous time steps.
  - f_t represents the importance of each dimension in c_{t-1}.
- If all values of k_t are 1, and all values of f_t are 0, then $c_t = i_t$.
  - New information i_t replaces old information $c_t - 1$, which is “forgotten”.

An Intuitive Interpetation

$$i_t = \sigma(W_i x_t + U_i z_{t-1} + B_i)$$
$$k_t = \sigma(W_k x_t + U_k z_{t-1} + B_k)$$
$$f_t = \sigma\left(W_f x_t + U_f z_{t-1} + B_f\right)$$
$$c_t = i_t * k_t + c_{t-1} * f_t$$

- There is a somewhat intuitive interpretation that motivated these formulas. According to this interpretation:
  - i_t represents new information, computed at time t.
  - k_t represents the importance of each dimension in i_t.
  - c_{t-1} represents old information, computed from previous time steps.
  - f_t represents the importance of each dimension in c_{t-1}.
- If all values of k_t are 0, and all values of f_t are 1, then $c_t = c_{t-1}$.
  - New information i_t is ignored, old information $c_t - 1$ is retained in c_t.

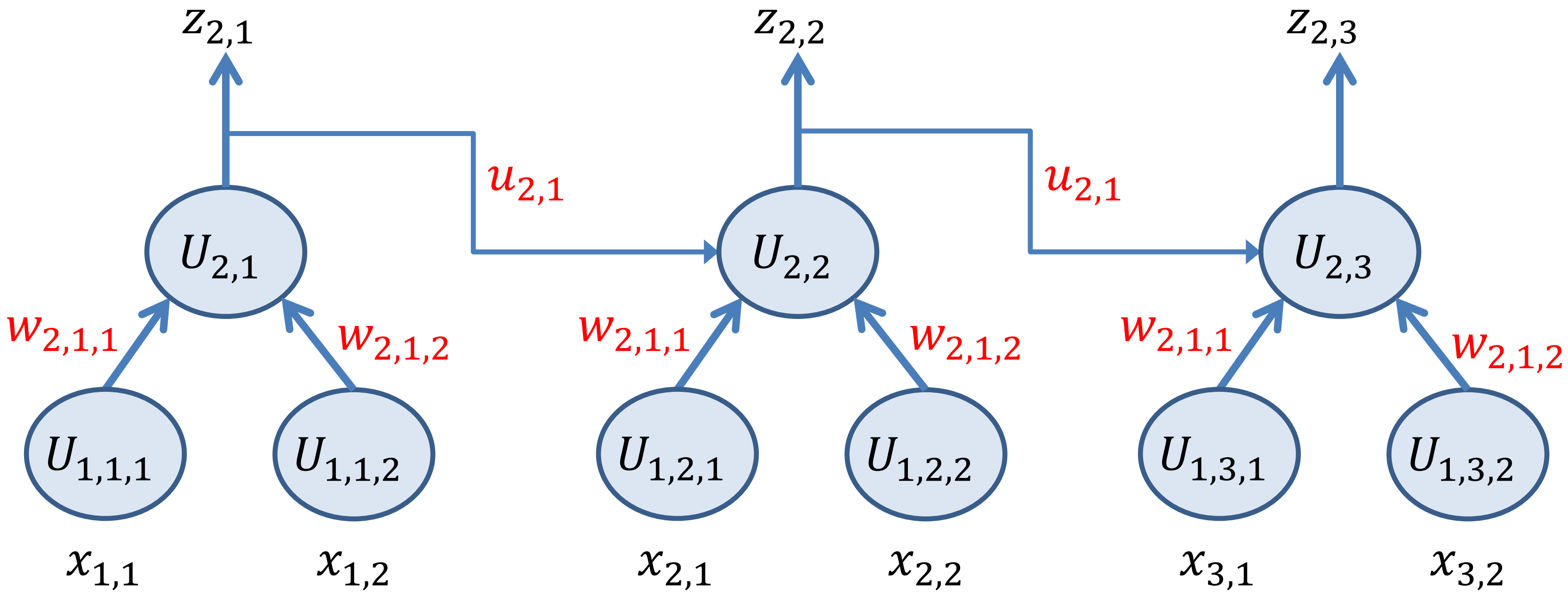
An Intuitive Interpetation

$$
\begin{aligned}
&i_t = \sigma(W_i x_t + U_i z_{t-1} + B_i)\\
&k_t = \sigma(W_k x_t + U_k z_{t-1} + B_k)\\
&f_t = \sigma\left(W_f x_t + U_f z_{t-1} + B_f\right)\\
&c_t = i_t * k_t + c_{t-1} * f_t
\end{aligned}
$$

- In the typical case, individual values of k_t and f_t will range between 0 and 1.
- Then, each dimension of vector c_t will be a weighted sum of:
  - new information from the corresponding dimension of i_t, with weight specified by the corresponding dimension of k_t.
  - old information from the corresponding dimension of c_{t-1}, with weight specified by the corresponding dimension of f_t.

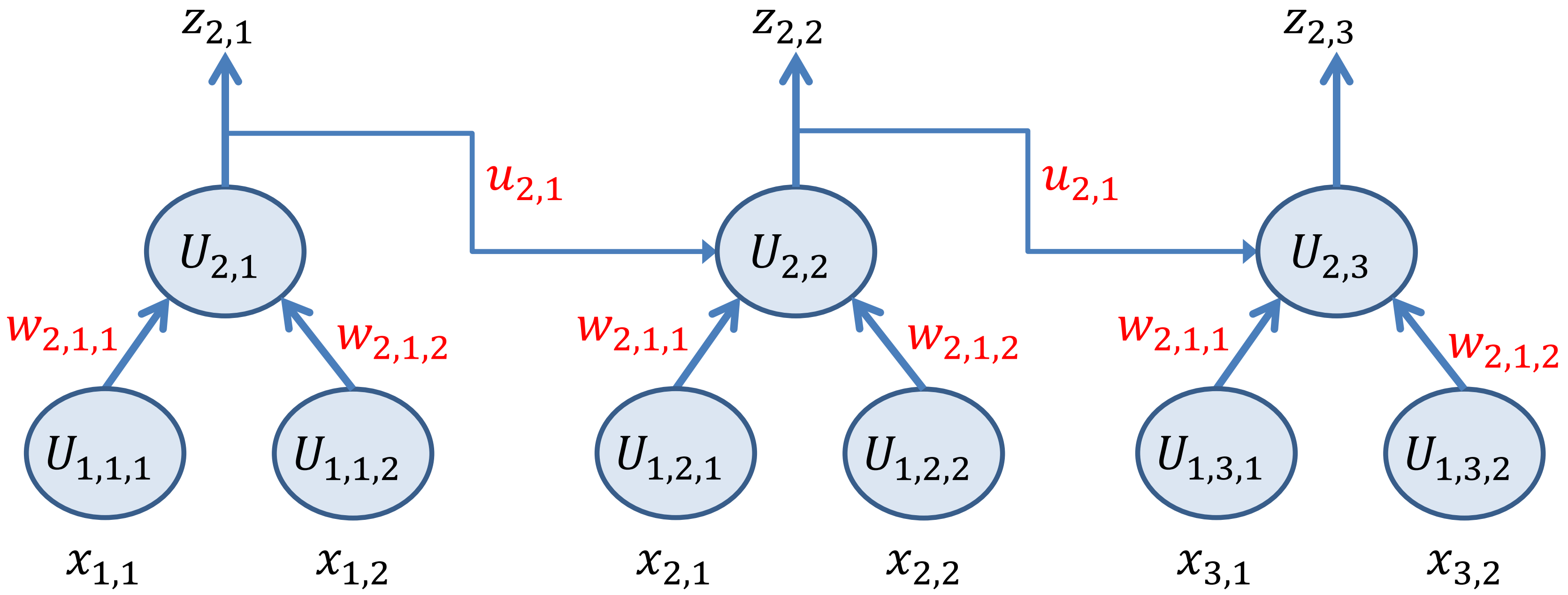
The Vanishing Gradient Problem

- An additional justification for the LSTM architecture is the "vanishing gradient" problem.
  - Under some neural network architectures, some weights do not get sufficiently updated during backpropagation, due to very small gradients.
  - Consequently, backpropagation does not learn good values for those weights.



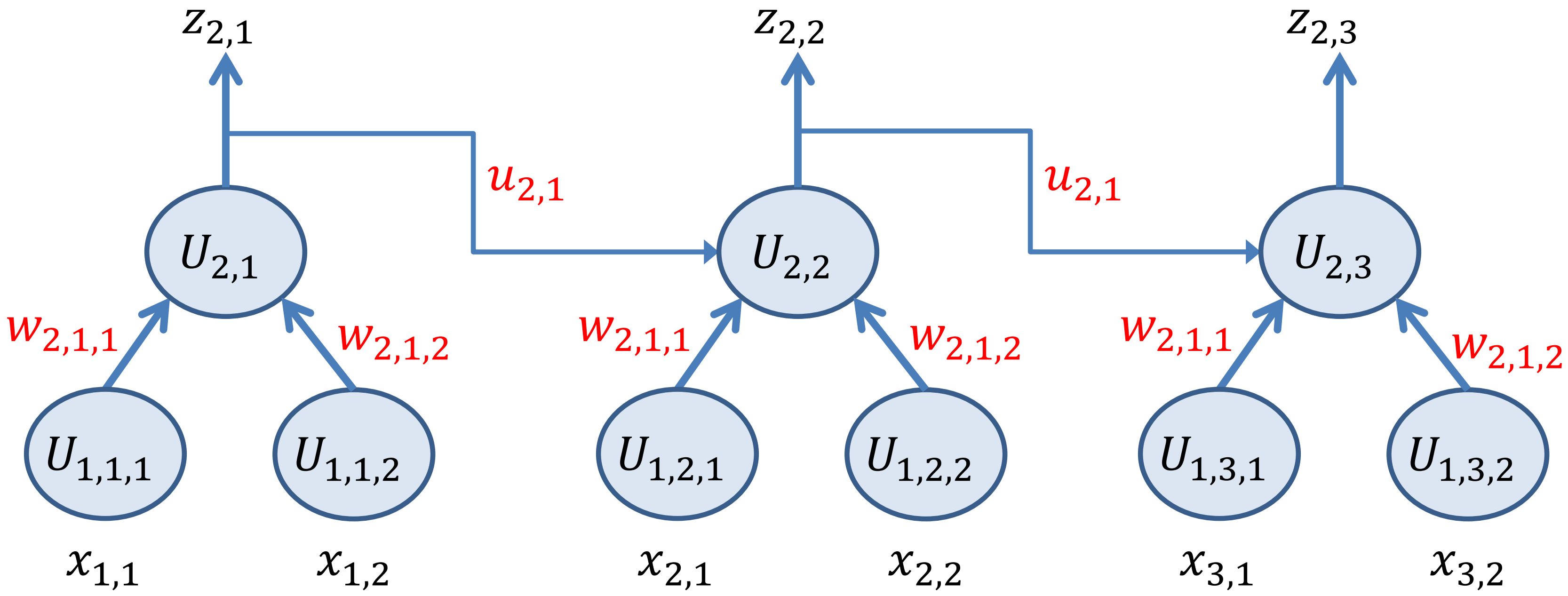
The Vanishing Gradient Problem

- To understand the problem, consider the toy RNN model below.
  - Let's assume that the only output of the model is $z_{2,3}$, so that the model estimates a single number.
  - Consider how weight $w_{2,1,1}$ (as an example) gets updated during training.



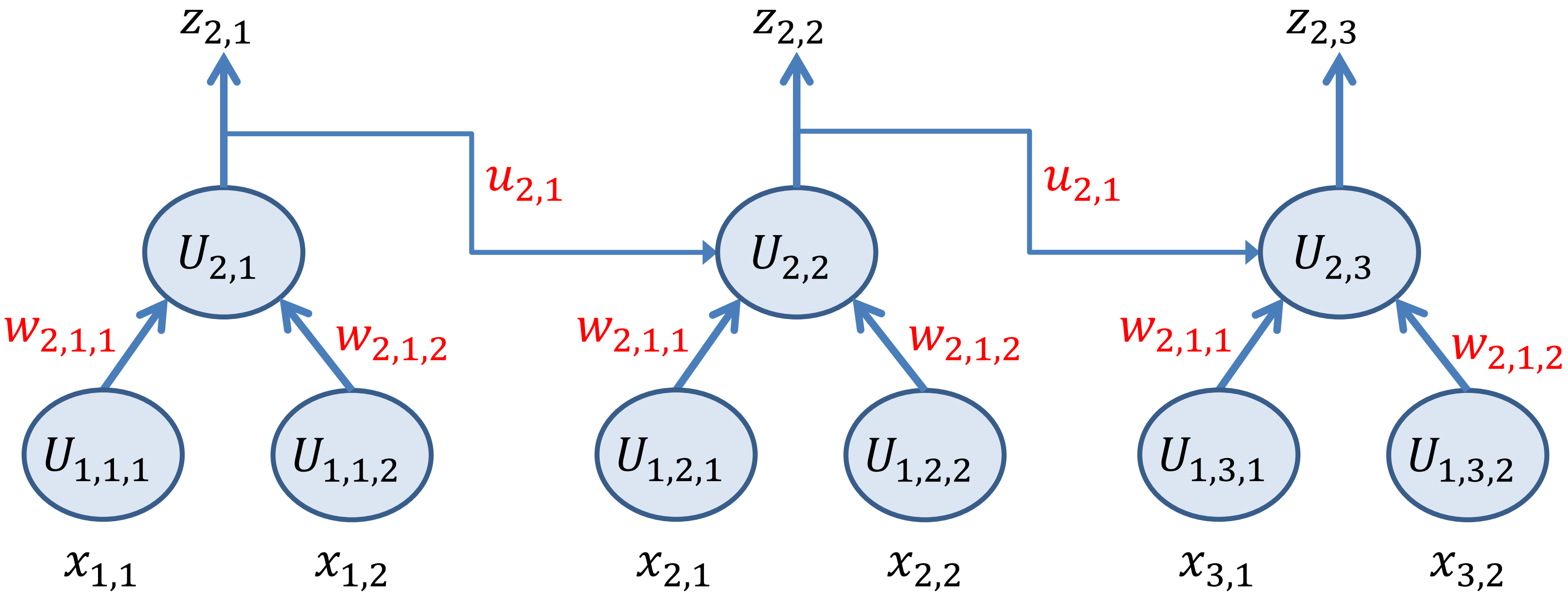
The Vanishing Gradient Problem

- Weight $w_{2,1,1}$ influences the output in multiple ways.
  - It gets multiplied by input $x_{1,1}$ during the first time step.
  - It gets multiplied by input $x_{2,1}$ during the second time step.
  - It gets multiplied by input $x_{3,1}$ during the third time step.



The Vanishing Gradient Problem

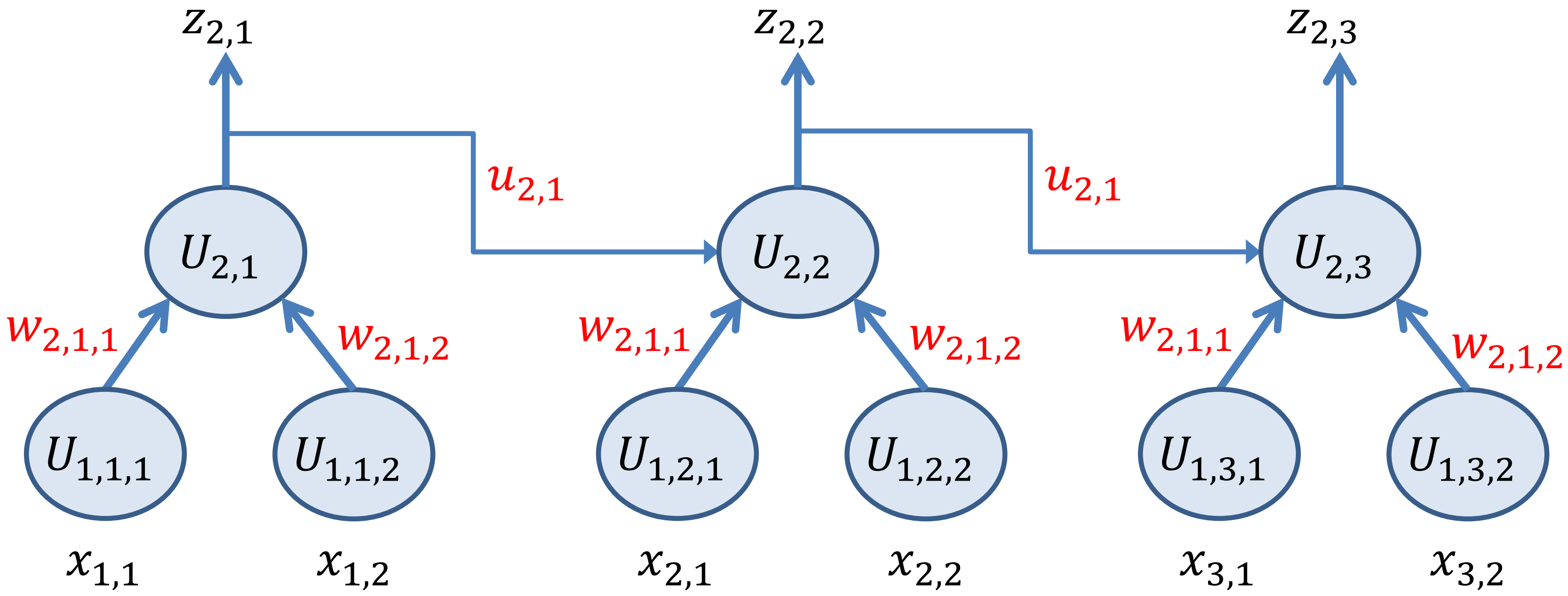
- During backpropagation, we update $w_{2,1,1}$ based on the three different ways in which it influenced the output.
- However, the third time step will often influence disproportionately how $w_{2,1,1}$ is updated.
- Why?



The Vanishing Gradient Problem

$$\frac{\partial E}{\partial w_{2,1,1}} = \frac{\partial E}{\partial z_{2,3}} \frac{\partial z_{2,3}}{\partial a_{2,3}} \frac{\partial a_{2,3}}{\partial w_{2,1,1}}$$

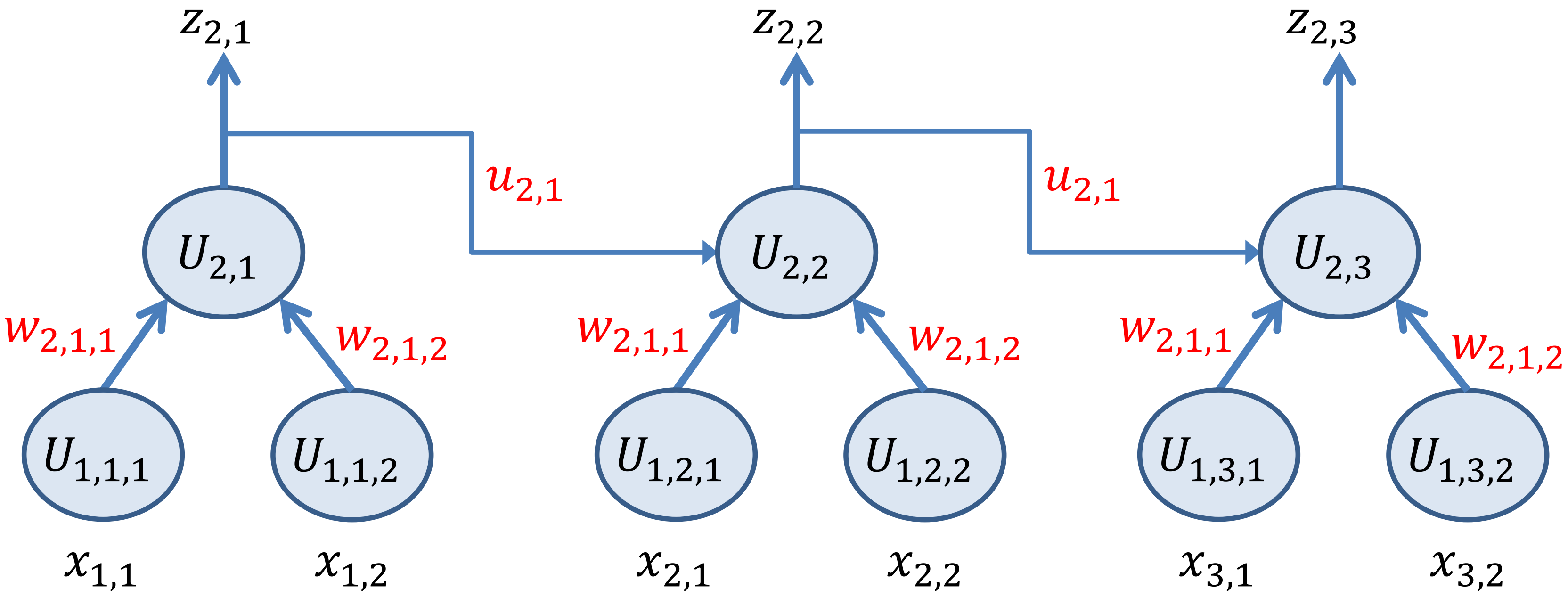
- We start by applying the chain rule to compute the partial derivative of the loss E with respect to $w_{2,1,1}$.



The Vanishing Gradient Problem

$$\frac{\partial a_{2,3}}{\partial w_{2,1,1}} = x_{3,1} + \frac{\partial a_{2,3}}{\partial z_{2,2}} \frac{\partial z_{2,2}}{\partial a_{2,2}} \frac{\partial a_{2,2}}{\partial w_{2,1,1}}$$

$$= x_{3,1} + \frac{\partial a_{2,3}}{\partial z_{2,2}} \frac{\partial z_{2,2}}{\partial a_{2,2}} \left(x_{2,1} + \frac{\partial a_{2,2}}{\partial z_{2,1}} \frac{\partial z_{2,1}}{\partial a_{2,1}} \frac{\partial a_{2,1}}{\partial w_{2,1,1}} \right)$$



The Vanishing Gradient Problem

- Combining the calculations from the previous slides we get:

$$\frac{\partial E}{\partial w_{2,1,1}} = \frac{\partial E}{\partial z_{2,3}} \frac{\partial z_{2,3}}{\partial a_{2,3}} x_{3,1} +$$

Influence of 3rd time step: product of 3 numbers.

$$\frac{\partial E}{\partial z_{2,3}} \frac{\partial z_{2,3}}{\partial a_{2,3}} \frac{\partial a_{2,3}}{\partial z_{2,2}} \frac{\partial z_{2,2}}{\partial a_{2,2}} x_{2,1} +$$

Influence of 2nd time step: product of 5 numbers.

$$\frac{\partial E}{\partial z_{2,3}} \frac{\partial z_{2,3}}{\partial a_{2,3}} \frac{\partial a_{2,3}}{\partial z_{2,2}} \frac{\partial z_{2,2}}{\partial a_{2,2}} \frac{\partial a_{2,2}}{\partial z_{2,1}} \frac{\partial z_{2,1}}{\partial a_{2,1}} x_{1,1}$$

Influence of 3rd time step: product of 7 numbers.

- The influence of each step is the product of many terms, which typically are between 0 and 1.

The Vanishing Gradient Problem

- We can extrapolate the previous formula to our temperature forecasting RNN.
- There, the input length is 120 steps.
- For any weight w_i connecting the SimpleRNN layer to the input, partial derivative $\frac{\partial E}{\partial w_i}$ will be a sum of 120 terms.
- Each term of $\frac{\partial E}{\partial w_i}$ will correspond to the influence of a single time step.
  - The influence of time step 120 will be a product of 3 numbers.
  - The influence of time step 119 will be a product of 5 numbers.
  - The influence of time step 118 will be a product of 7 numbers.

…

  - The influence of time step 1 will be a product of 241 numbers.

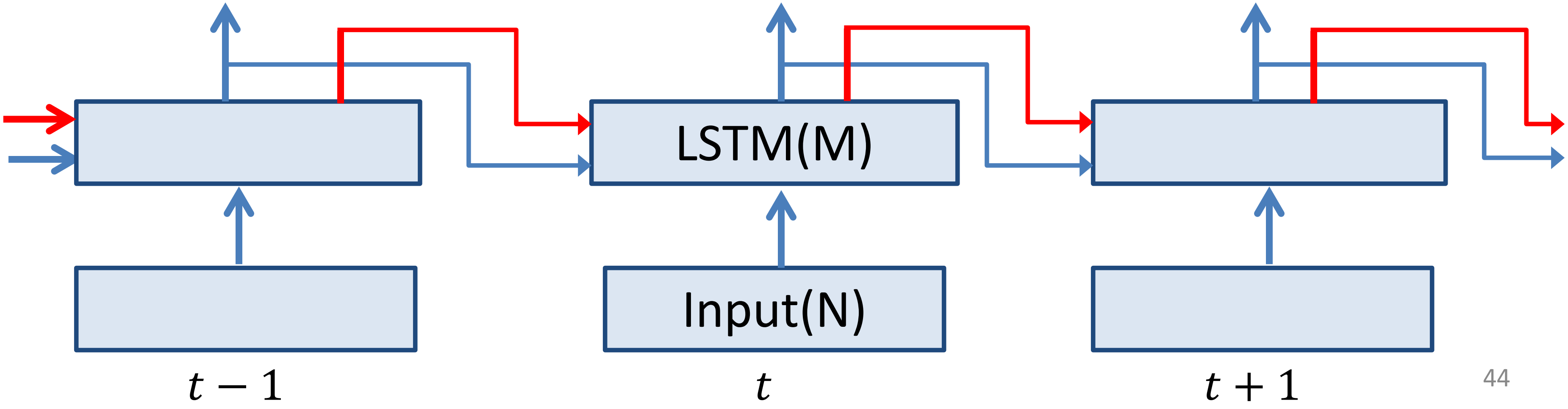
The Vanishing Gradient Problem

- Each term of $\frac{\partial E}{\partial w_i}$ will correspond to the influence of a single time step.
  - The influence of time step 120 will be a product of 3 numbers.
  - The influence of time step 119 will be a product of 5 numbers.
  - The influence of time step 118 will be a product of 7 numbers.

  …
  - The influence of time step 1 will be a product of 241 numbers.
- So, the influence of time step 1 will be a product of 241 numbers, which will usually be between 0 and 1.
- This will be a very small quantity.
- Overall, the influence of a time step drops exponentially as we move from the end towards the beginning of the input time series.

LSTMs and Vanishing Gradients

- The carry output can (potentially) remember information from earlier time steps.
- This allows calculations from earlier time steps to influence the output more heavily than in a SimpleRNN layer.
  - Influencing the output more heavily means higher contributions to the partial derivatives of weights.
  - That way, the model can learn to give more importance to earlier time steps.



Detour: ResNet

- The vanishing gradient problem is not particular to RNNs.
- Any deep network involves a sequence of calculations, mapping inputs to outputs.
  - Calculations earlier in the sequence end up making smaller contributions to partial derivatives of weights.
- For convolutional neural networks (CNNs), a popular method for resolving the vanishing gradient problem is ResNet.
- We will not discuss ResNet in this class.
  - The method is somewhat similar to LSTM, by providing a way for earlier calculations to be "remembered" in later layers.
- If you are interested in learning more about ResNet, a good starting point is the Wikipedia article:

https://en.wikipedia.org/wiki/Residual_neural_network

GRU

- GRU stands for Gated Recurrent Unit.
- GRU layers are yet another type of recurrent layer.
- GRU layers can be used instead of SimpleRNN or LSTM.
- You can think of a GRU layer as an approach that is more complicated than a SimpleRNN layer and more simple than an LSTM layer.
- We will not discuss GRU layers any further in this class.
- As usual, the Wikipedia article is a good starting point for more info:

https://en.wikipedia.org/wiki/Gated_recurrent_unit

Recurrent Dropout

- Dropout can be used with recurrent layers (such as SimpleRNN, LSTM, GRU).
- However, the picture is more complicated, because the same weights are used in multiple time steps.
- In practice, better results are usually obtained if the same weights are “dropped” at each time step.
- A normal Keras dropout layer does not know how to do that.
- To use dropout properly with recurrent layers, you should use the optional parameters **dropout** and **recurrent_dropout**.

Recurrent Dropout

```
model = keras.Sequential([keras.Input(shape=input_shape),
                          keras.layers.LSTM(32, dropout=0.3, recurrent_dropout=0.25),
                          keras.layers.Dropout(0.5),
                          keras.layers.Dense(1),])
```

- This piece of code shows an example of how to combine different dropouts.
- The LSTM layer specifies a **dropout** value of 0.3.
  - This means that 30% of the weights between the input layer and the LSTM layer will be dropped for each training object.
- The LSTM layer specifies a **recurrent_dropout** value of 0.25.
  - This means that 25% of the weights applied to outputs and carry values from the previous time step will be dropped for each training object.

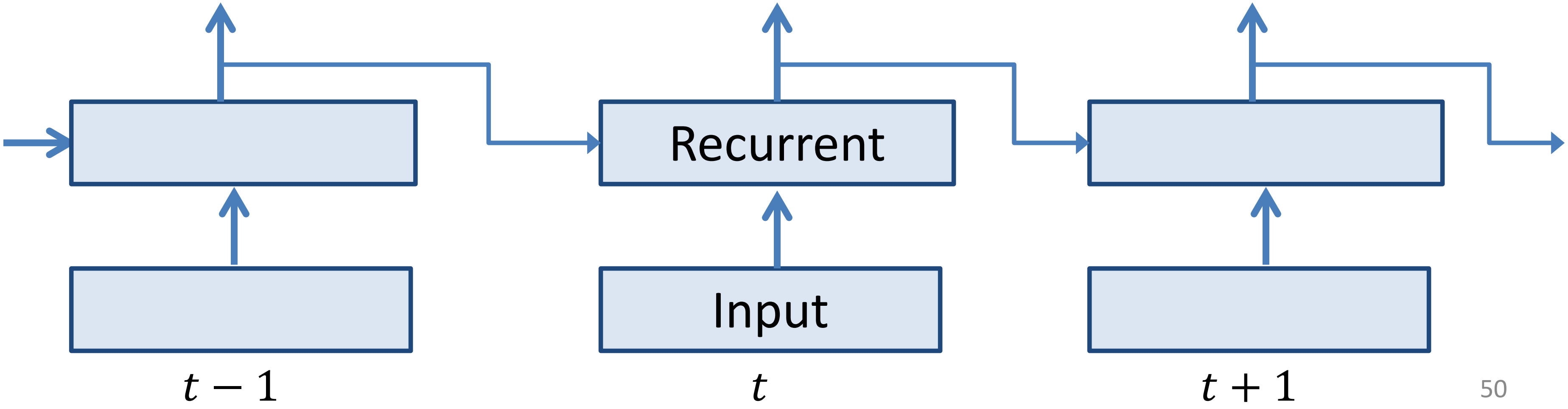
Recurrent Dropout

```
model = keras.Sequential([keras.Input(shape=input_shape),
                          keras.layers.LSTM(32, dropout=0.3, recurrent_dropout=0.25),
                          keras.layers.Dropout(0.5),
                          keras.layers.Dense(1),])
```

- Notice that we still use a regular Keras dropout layer between the LSTM layer and the fully connected output layer.
  - Here we specify that, for each training object, 50% of the weights connecting the LSTM outputs to the Dense layer will be dropped.
- Optional parameters **dropout** and **recurrent_dropout** specify how to do dropout of weights **incoming** to the recurrent layer.
- For weights **outgoing** from the layer and incoming to a fully connected layer, a regular dropout layer should be used.

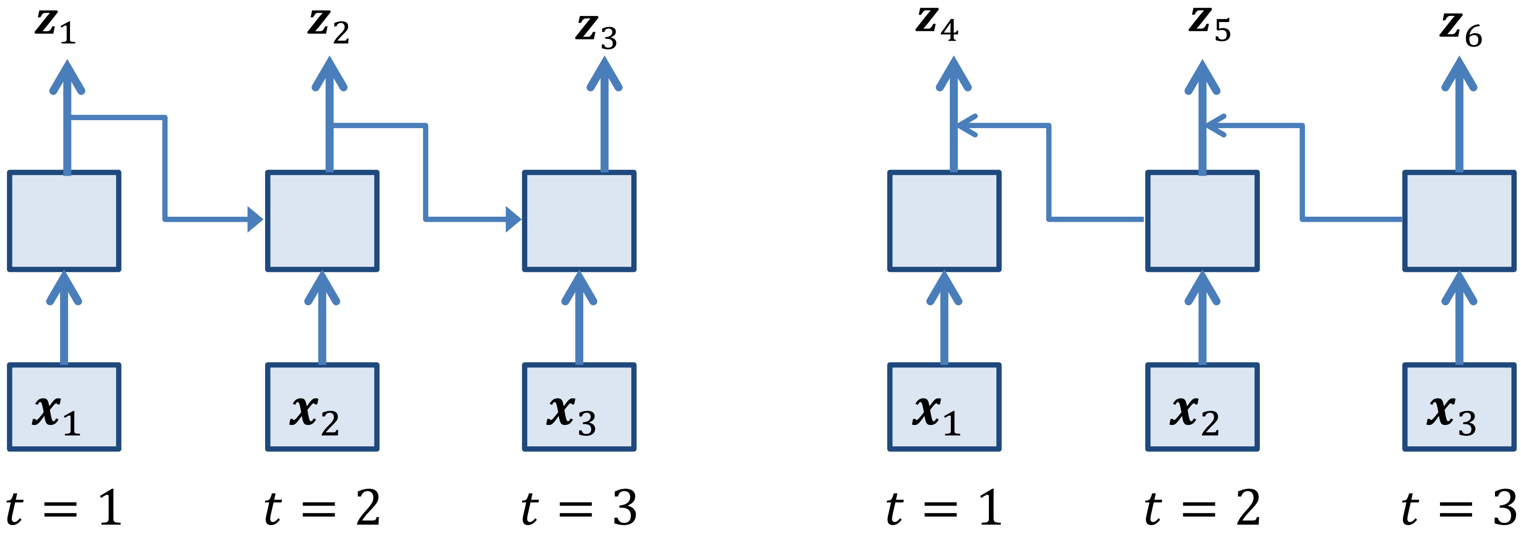
Bidirectional Layers

- A recurrent layer processes information from time step to time step, in chronological order.
- Would it make a difference if information was processed in reverse chronological order?
  - It might.
- How can we know which order is better?
  - We usually don't.

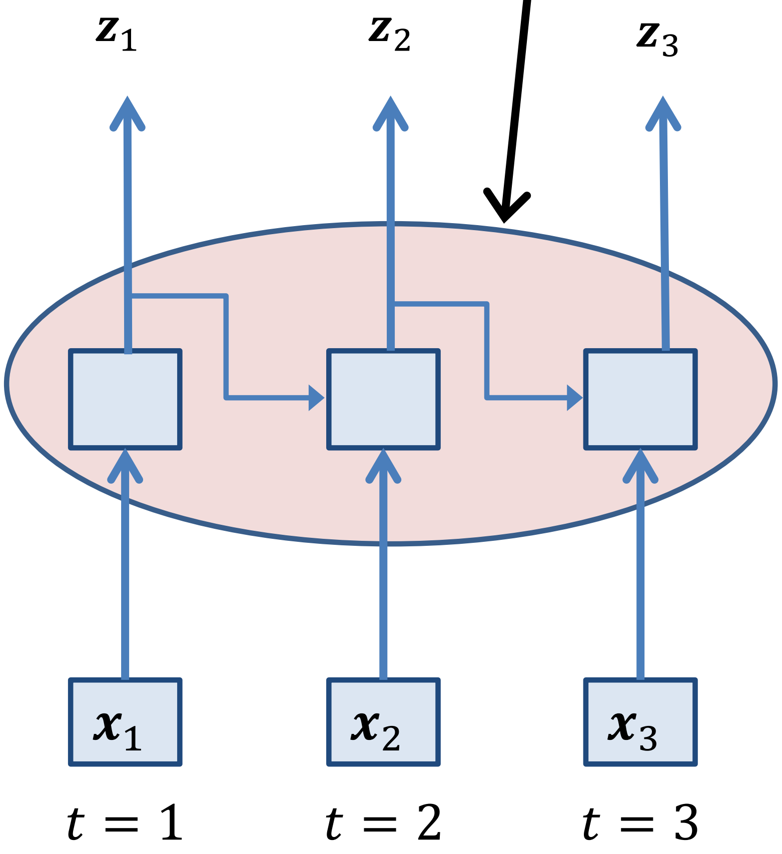


Bidirectional Layers

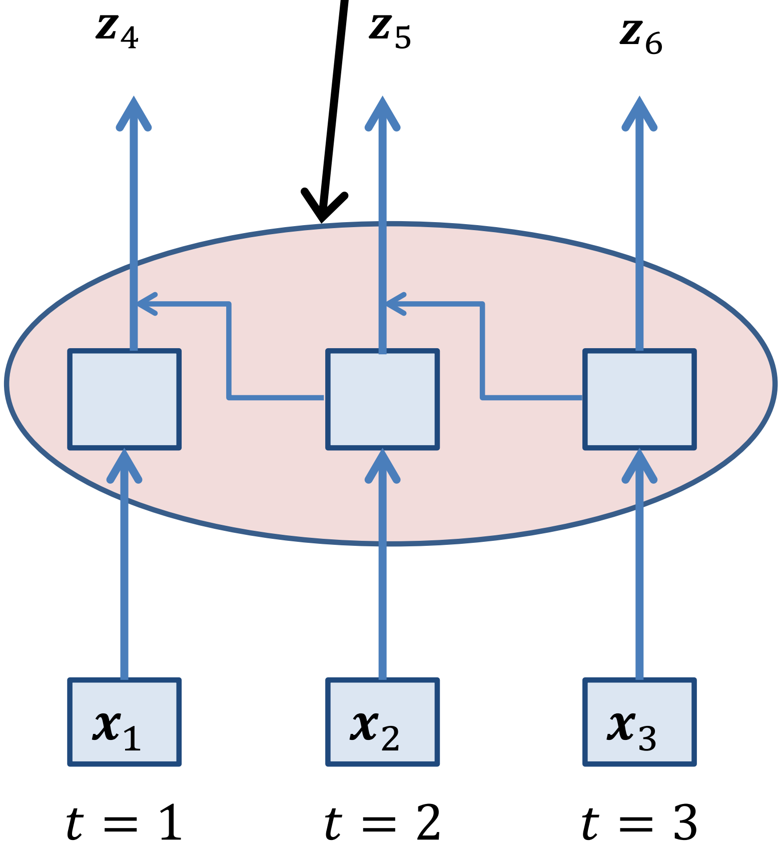
- A bidirectional layer processes information in both chronological and anti-chronological order.
- Essentially, a bidirectional layer consists of two recurrent layers, each processing information in different order.
- The output of the bidirectional layer is simply the merged output of both layers.



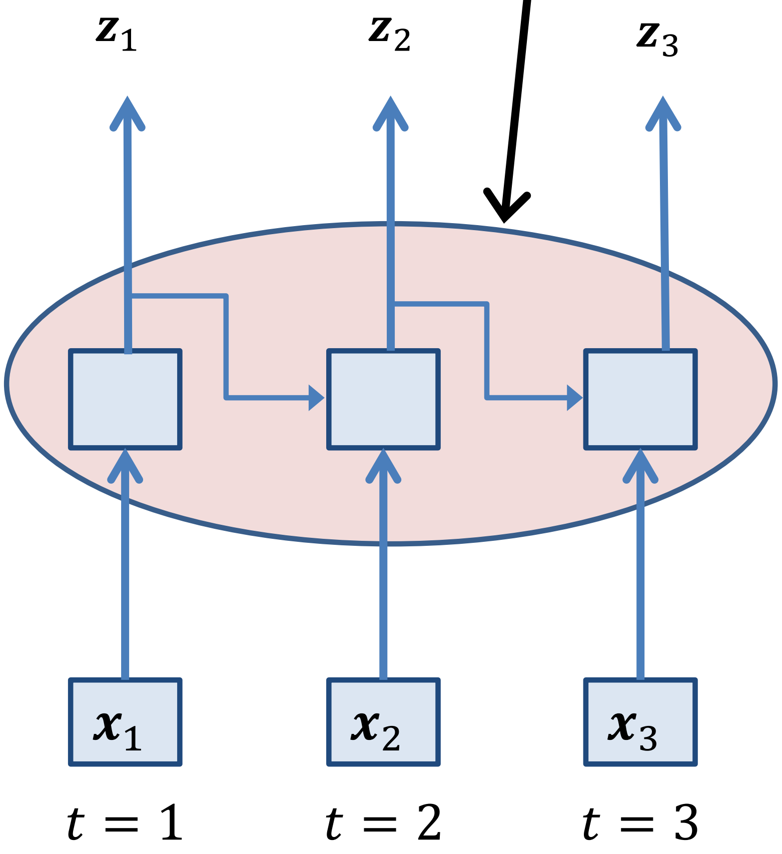
Recurrent layer (SimpleRNN, LSTM, GRU) processing information in chronological order.



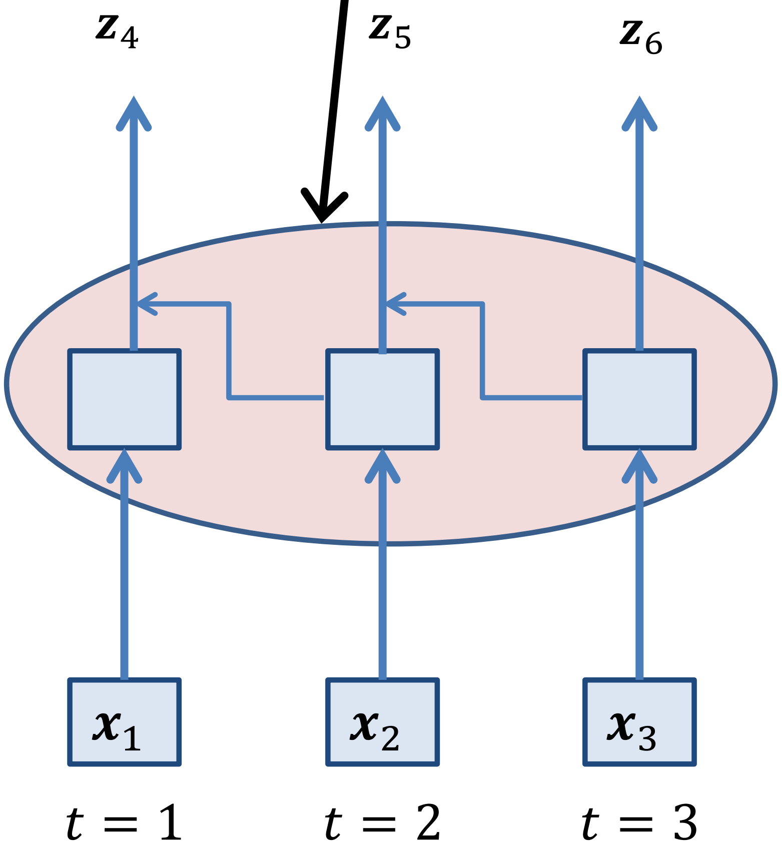
Recurrent layer (SimpleRNN, LSTM, GRU) processing information in **REVERSE** chronological order.



Recurrent layer (SimpleRNN, LSTM, GRU) processing information in chronological order.

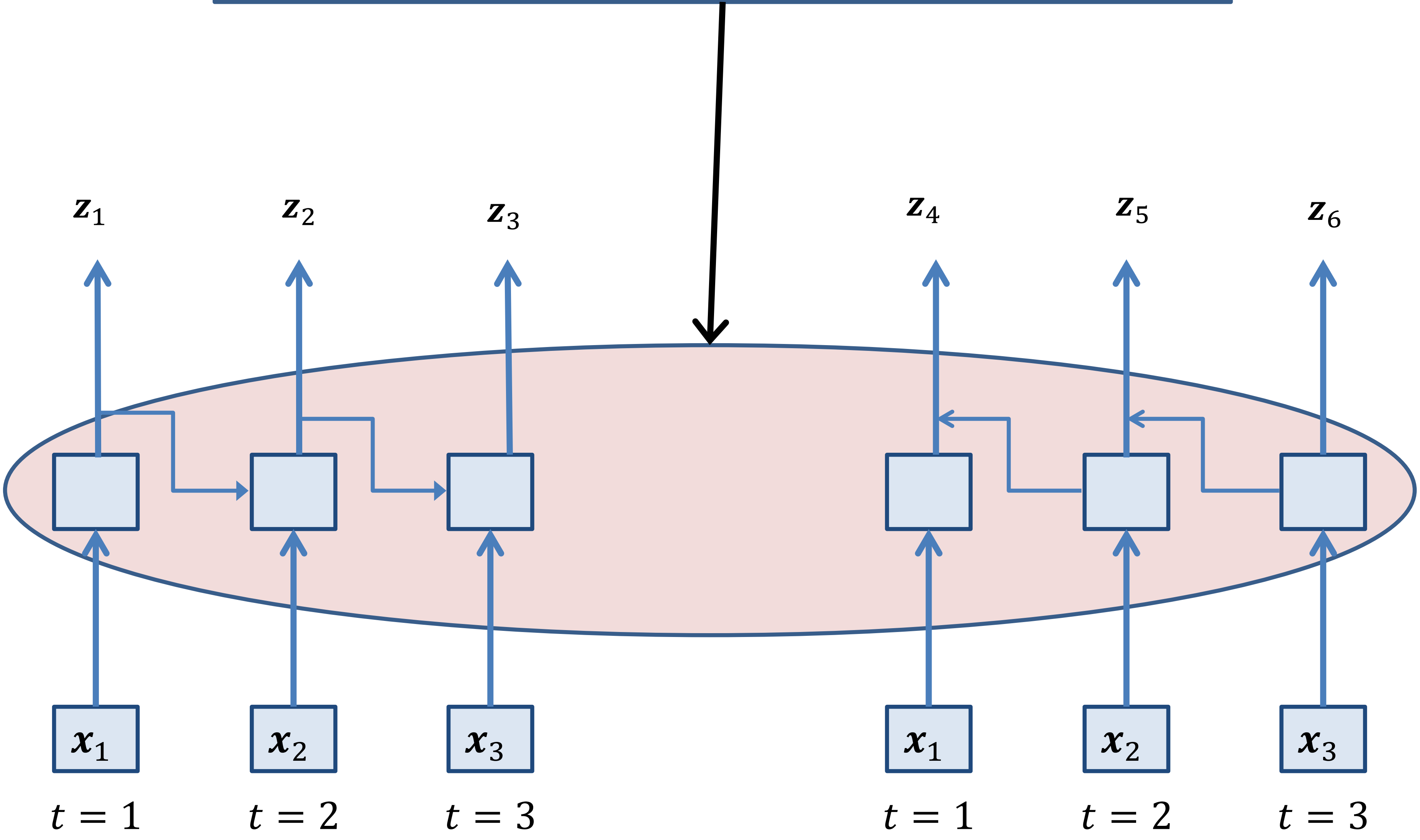


Recurrent layer (SimpleRNN, LSTM, GRU) processing information in **REVERSE** chronological order.

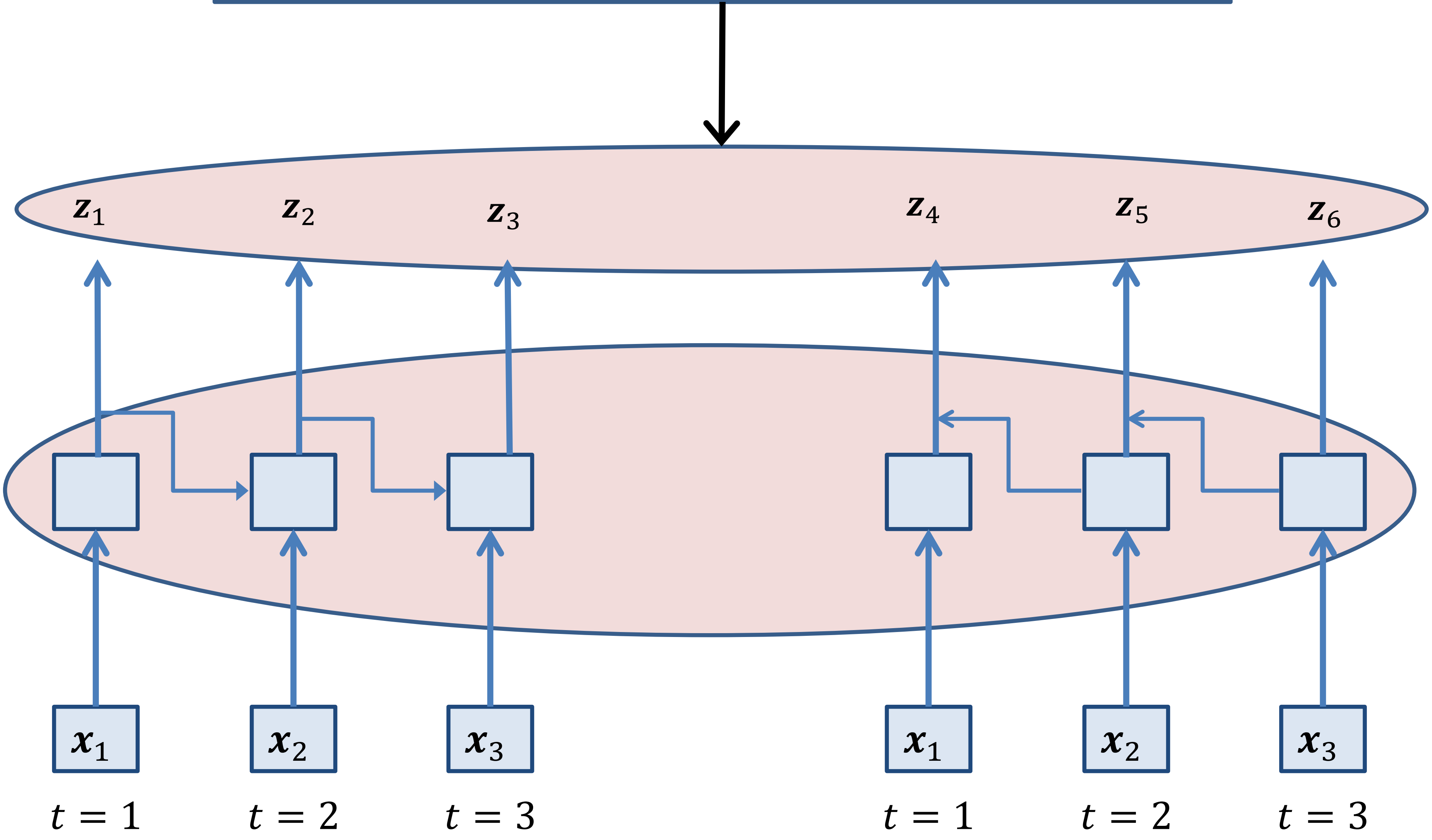


Both layers receive the exact same inputs.

These two recurrent layers, combined, form what we call a **bidirectional layer**.



The output of the bidirectional layer is the concatenation of the outputs of the two recurrent layers.



Bidirectional Layers in Keras

```
model = keras.Sequential([keras.Input(shape=input_shape),
                    keras.layers.Bidirectional(keras.layers.LSTM(32)),
                    keras.layers.Dense(1),])
```

- The Bidirectional layer takes as argument a recurrent layer.
- The Bidirectional layer creates two replicas of the recurrent layer.
  - The first replica processes information in chronological order.
  - The second replica processes information in antichronological order.
- The output of the Bidirectional layer is the concatenated output of the two replicas.

RNN Summary

- Recurrent layers process information one time step at a time.
- The most simple recurrent layer is SimpleRNN.
- The LSTM layer is a more complicated recurrent layer, that allows the model to learn when to remember old information and when to replace that “memory” with new information.
- Recurrent dropout must be handled differently than regular dropout, using optional parameters when creating the recurrent layers.
  - **dropout** for weights connecting the previous layer to the recurrent layer.
  - **recurrent_dropout** for weights connecting outputs of the recurrent layer from previous time to the current time step.
- Bidirectional layers allow processing information both in chronological and in antichronological order.