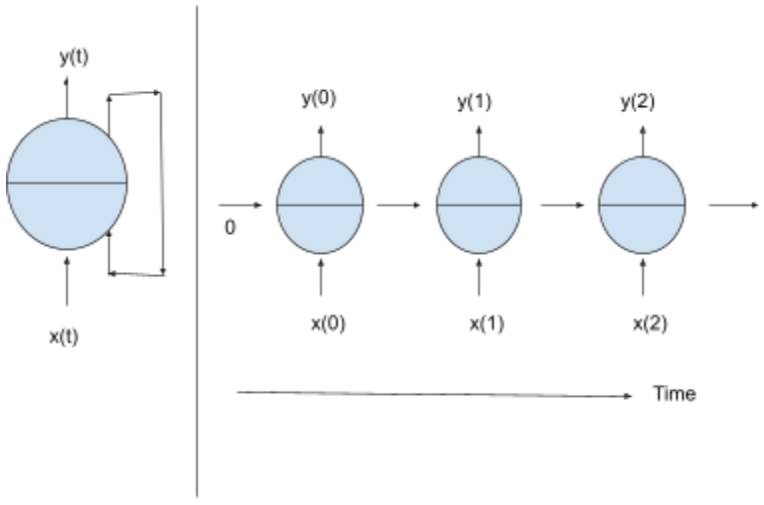
Processing Sequences Using RNNs and CNNs

Recurrent Neural Networks

- Recurrent neural networks (RNNs) can predict the future up to a point.
    - They can analyze time series data, learn patterns, and use it to forecast the future.
    - RNNS can work on sequences of arbitrary lengths, such as sentences, documents, or audio samples.
    - They are helpful for NLP applications, such as automatic translation or speech-to-text.

Recurrent Neurons and Layers

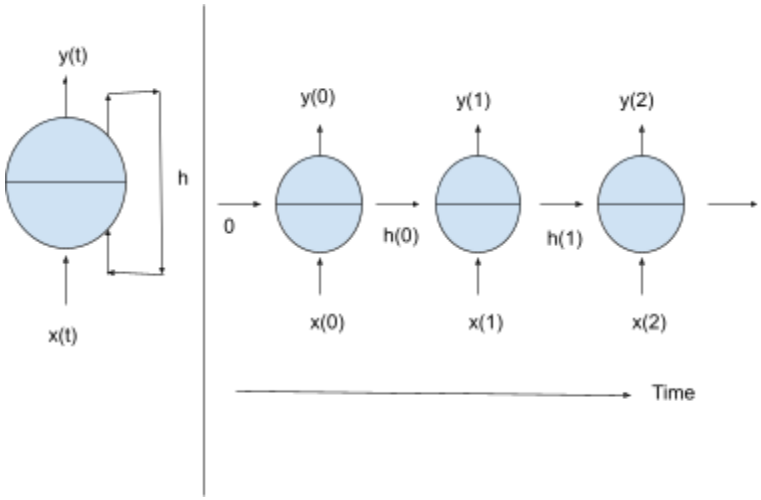
- An RNN looks similar to a feedforward neural network but has backward connections.
    - At each time step t, a recurrent neuron receives the inputs x(t) and its output from the previous step ŷ(t-1).
    - Since there is no previous output at the first time step (t = 0), it is generally set to 0.
    - We can represent an RNN against the time axis, called unrolling the network through time.



    - Each recurrent neuron has two sets of weights: one for the inputs x(t) and one for the outputs of the previous time step ŷ(t-1).
    - The output formula is $Y(t) = \phi X(t)Wx + Y(t-1)Wy + b$ where
        - Y(t) is an m x $n_{neurons}$ matrix containing the layer's outputs at time step t for each instance in the mini-batch (m is the number of instances).
        - X(t) is an m x n_{inputs} matrix containing the inputs for all instances.

- Wx is an n_{inputs} x $n_{neurons}$ matrix containing the connection weights for the inputs of the current time step.
- Wy is an $n_{neurons}$ x $n_{neurons}$ matrix containing the connection weights for the outputs of the previous time step.
- b is a vector of size $n_{neurons}$ containing each neuron's bias term.

Memory Cells

- A part of a neural network that preserves some state across time steps is called a memory cell.
- A cell's state at time step t is a function h(t) of some inputs at that time step and its state of the previous time step.
  - The h stands for hidden. The function is $h(t) = f(x(t), h(t-1))$.
  - 

Input and Output Sequences

- In a sequence-to-sequence network, the RNN can simultaneously take a sequence of inputs and produce a sequence of outputs.
  - It is useful for forecasting time series, such as a home's daily power consumption.
  - We feed the network data over the last N days and train it to output the power consumption from N - 1 days ago to tomorrow.
- In a vector-to-sequence network, we feed the network the same input vector at each time step and let it output a sequence.
  - For example, the input could be an image, and the output could be a caption for that image.
- We can have a sequence-to-vector network, called an encoder, followed by a vector-to-sequence network, called a decoder.
  - For example, This network could translate a sentence from one language to another.
  - We would feed the network a sentence in one language, the encoder would convert this sentence into a single vector representation, and the decoder would decode this vector into a sentence in another language.

Training RNNs

- We unroll the RNN through time and use regular backpropagation to train it. This strategy is called backpropagation through time (BPTT).
    - There is a forward pass through the unrolled network, and the output sequence is evaluated using a loss function.
        - The loss function's gradients are propagated backward through the unrolled network.
        - The loss function is $\mathcal{L}(Y(0), Y(1),\ldots, Y(T); \hat{Y}(0), \hat{Y}(1),\ldots, \hat{Y}(T))$ where Y(i) is the ith target, $\hat{Y}$(i) is the ith prediction, and T is the max time step.
        - Note that the loss function may ignore some outputs.
        - Lastly, BPTT performs a gradient descent to update the parameters.

Forecasting a Time Series

- First, we have to load the data. Then, we can visualize the data to search for any patterns.
- We can make a time series, data with values at different time steps, usually at regular intervals.
    - If there are multiple values per time step, this is called multivariate time series. If there is only one value, this is called univariate time series.
    - Forecasting, or predicting future values, is the most common task when working with time series.
- Many time series have patterns called seasonality, spanning weeks, months, or years.
    - When the seasonality is prominent, we can copy past values, which yields reasonably good results. This is called naive forecasting.
- We can compute the difference between the time series at one point in time and the time series at another. This is called differencing.
    - Differencing helps remove trend and seasonality from a time series.
    - Studying a stationary time series with statistical properties that remain constant over time is easier.
- When a time series is correlated with a lagged version of itself, it is autocorrelated.
- The mean absolute percentage error (MAPE) is a helpful metric for evaluating the time series model.
    - Note that variables with more instances will have more errors, so MAPE considers this to output its value.

The ARMA Model Family

- The autoregressive moving average (ARMA) model computes its forecasts using a weighted sum of lagged values and corrects them by adding a moving average.
    - The ARMA model has two hyperparameters: p, which controls how far back into the past the model should look, and q, the moving average component.

- This model assumes that the time series is stationary. If not, differencing can help.
    - Differencing transforms linear and quadratic trends to constant values.
    - Running d consecutive rounds of differencing computes an approximation of the dth-order derivative of the time series.
    - The hyperparameter d is called the order of integration.
- The autoregressive integrated moving average (ARIMA) is a variant of the ARMA model.
    - This model runs d rounds of differencing to make the time series more stationary, and then it applies a regular ARMA model.
    - For forecasts, it uses the ARMA model and then adds back the terms subtracted by differencing.
- Another ARMA variant is the seasonal ARIMA (SARIMA) model.
    - It models similarly to the ARIMA model but also models a seasonal component for a given frequency (e.g., weekly) using the ARIMA approach.
    - It has seven hyperparameters: p, d, q (same as ARIMA), P, D, and Q to model the seasonal pattern and s, the period of the seasonal pattern.
- The ARIMA model is available through the **statsmodel.tsa.arima.model** package.
    - The **asfreq()** method defines the frequency: “D” for daily, “W” for weekly, etc.
    - The *order* argument specifies the p, d, and q hyperparameters.
    - The *seasonal_order* argument specifies the P, D, Q, and s hyperparameters.
- To find good hyperparameters for the SARIMA model, we can use the brute-force grid search approach.
    - The p, q, P, and Q hyperparameters usually range from 0 to 2, sometimes up to 5 or 6.
    - The d and D hyperparameters are typically 0 or 1, sometimes 2.
    - The s hyperparameter depends on the analysis frequency (e.g., 7 for weekly).

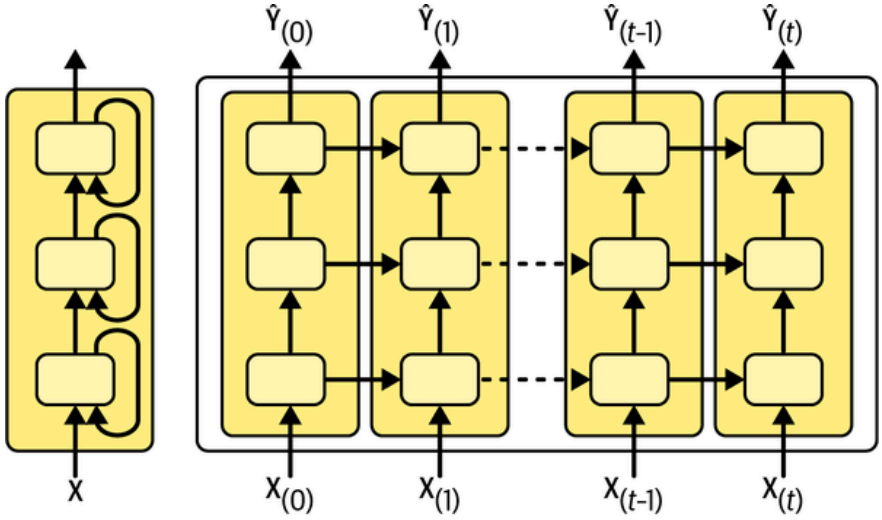
Preparing the Data for Machine Learning Models

- For a simple linear model, the input must be sequences containing N instances (for N days).
    - For each input sequence, the model will output the forecast for time step t + 1.
    - We can use the **tf.keras.utils.timeseries_dataset_from_array()** function to prepare the data.
    - We can also use the **window()** method, which returns a nested dataset (a dataset of window datasets).
        - This is useful when we want to transform each window through the dataset methods (shuffle, batch, etc.)
    - The model accepts tensors, not nested datasets. We can use the **flat_map()** method to flatten the dataset.
        - For example, if we have [[1, 2], [3, 4, 5, 6]], the flattened dataset will contain [1, 2, 3, 4, 5, 6].
    - Then, we split the dataset into the training, validation, and test sets. For time series, we generally want to split across time.
    - For this model, the Huber loss may work better than MAE.

Forecasting Using a Simple RNN

- We can build a recurrent layer using Keras's **SimpleRNN()** layer.
    - Note that all recurrent layers in Keras expect 3D inputs of shape [batch size, time steps, dimensionality] where dimensionality is 1 for univariate time series.
    - The *input_shape* argument ignores the first dimension.
    - Setting the second dimension to *None* tells the layer that it can use any size.
- A model with an output layer with one neuron is a sequence-to-vector model. The output layer typically does not use an activation function.

Forecasting Using a Deep RNN

- A deep RNN is a stack of multiple layers of cells.
    - 

- We can use a stack of SimpleRNN() layers to build a deep RNN.

Forecasting Multivariate Time Series

- We can include multiple variables within the model, which will output predictions for both variables.
    - Unlike a univariate model, a multivariate model will receive multiple inputs instead of one.
    - Also, the multivariate model's output layer will have multiple neurons (one for each variable) instead of one.

Forecasting Several Time Steps Ahead

- We can predict the value several steps ahead by changing the targets appropriately.
    - In other words, instead of predicting ahead by one day, we can predict ahead by more.

    - One way to do this is to create a model that predicts the next value, add that value to the inputs, use the model to predict the next value, and so on.
    - Another approach is to train an RNN to predict the next values in one shot.
        - We can still use a sequence-to-vector model, but it will output several values instead of one.
        - The targets must be the vectors containing the next values.
        - We can use the **timeseries_dataset_from_array()** function but have it create a dataset with no targets and extend its sequence length.
        - The output layer will have multiple neurons instead of one.
        - Then, we can have the model predict the next values.

Forecasting Using a Sequence-to-Sequence Model

- Instead of training the model to forecast the next values only at the last time step, we can train it to forecast the next values at every time step.
    - In other words, we can turn a sequence-to-vector RNN into a sequence-to-sequence RNN.
    - The targets must be sequences of consecutive windows, shifted by one step at each time step. The **windows()** method may come in handy.
    - Then, we can create a custom class that builds the sequence-to-sequence model.

Handling Long Sequences

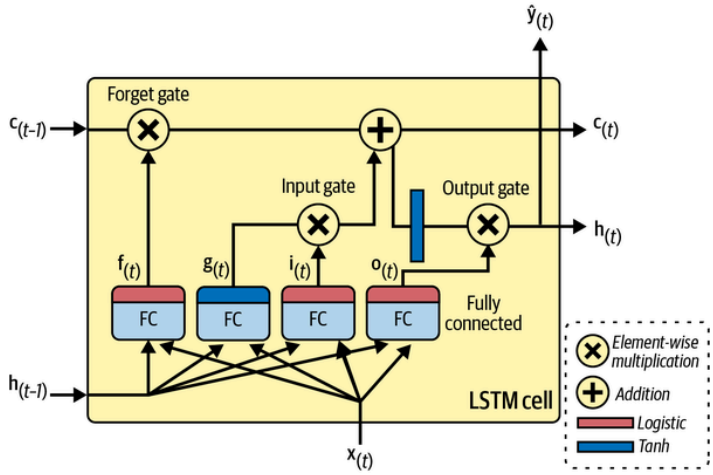
- Training an RNN on long sequences requires running it over many time steps.
- Also, the RNN may have unstable gradients, meaning that it will gradually forget the first inputs in the sequence.

Fighting the Unstable Gradients Problem

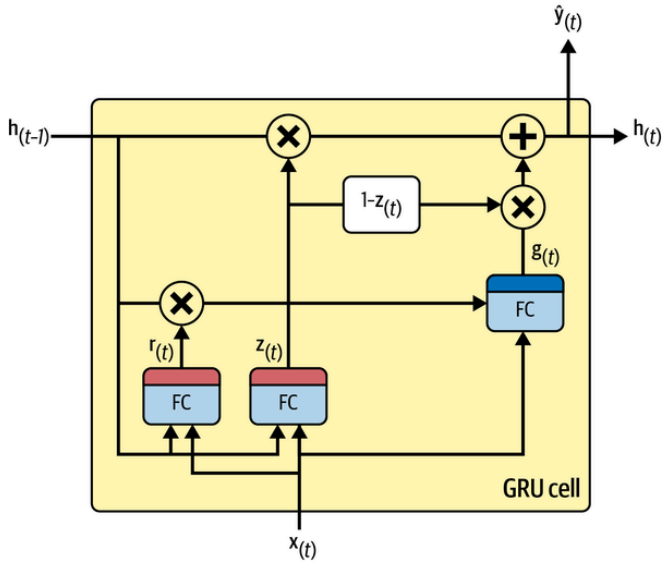
- The same techniques to fix this problem in DNNs can work for RNNs too: good parameter initialization, faster optimizers, dropout, etc.
    - Non-saturating activation functions (e.g., ReLU) may not help as much.
    - Batch Normalization cannot be used between time steps, only between recurrent layers.
    - BN can normalize across the batch dimension, but there is another type of normalization that is more useful for deep RNNs.
- Layer normalization normalizes across feature dimensions.
    - Layer normalization can compute the required statistics on the fly, at each time step, independently for each instance.
    - We can use Keras's **LayerNormalization()** layer.
- Most recurrent layers and cells in Keras have *dropout* and *recurrent_dropout* hyperparameters.
    - Dropout defines the dropout rate to apply to the inputs.

    - Recurrent dropout defines the dropout rate for the hidden states, between time steps.

Tackling the Short-Term Memory Problem

- After a while, the RNN's state contains no memory of the first inputs due to the data's transformations as it traverses the RNN.
- The long short-term memory (LSTM) cell can detect longer-term patterns in the data.
    - Keras provides the **LSTM()** layer that we can use instead of the SimpleRNN() layer.
    - The LSTM cell is split into two vectors: h(t) as the short-term state and c(t) as the long-term state.
    - 

    - The long-term state traverses the network from left to right.
        - First, it goes through a forget gate, dropping some memories, and the addition operation adds new memories from the input gate.
        - The resulting c(t) is sent straight out.
        - The network copies and passes the long-term state through the tanh function, and the output gate filters the result.
        - This produces the short-term state h(t), which is equal to the cell's output for the current time step y(t).
    - The network feeds the current input vector x(t) and previous short-term state h(t-1) to four fully connected layers.
        - The g(t) layer analyzes the current inputs x(t) and the previous short-term state h(t-1).
        - The three other layers are gate controllers. They produce values ranging from 0 to 1. If the output is 0, they close the gate and open it if it's 1.
- The gated recurrent unit (GRU) cells are simplified versions of the LSTM cells.



- Both state vectors are merged into vector h(t).
- Gate controller z(t) controls the forget gate and the input gate. If it outputs a 1, the forget gate is open and the input gate is closed.
- There is no output gate; instead, gate controller r(t) controls which part of the previous state will be shown to the main layer g(t).

- Even with LSTM and GRU cells, RNNs have limited short-term memory and struggle to learn long-term patterns in sequences of over 100 time steps.

Using 1D Convolutional Layers to Process Sequences

- We can use 1D convolutional layers to stabilize training.
  - Since the kernel size is larger than the stride, all inputs will compute the layer's output so that the model can learn to preserve only useful information.
  - The 1D convolutional layer is available through Keras's **Conv1D()** layer.

WaveNet

- A WaveNet is a neural network with stacked 1D convolutional layers and double the dilation rate at every layer.
  - The first convolutional layer glimpses two time steps ahead at a time, the second one sees four at a time, the third sees eight, and so on.
  - We can build a Sequential() model and add 1D convolutional layers in a for loop.
    - The 1D convolutional layers use "causal" padding, which appends zeros at the start of the input sequence instead of on both sides.
    - This ensures that the convolutional layer does not glimpse into the feature when making predictions.