Highly Integrated System Project

Assignment 4

Aaliya Javed
Zain Hanif

November 23, 2024

Deep Learning for Time Series Forecasting

In the previous chapter, we introduced deep learning broadly. This chapter focuses on its application in time series forecasting, a domain where deep learning methods have shown great promise.

We will cover the following topics:

- Encoder-decoder paradigm
- Feed-forward networks
- Recurrent neural networks (RNNs)
- Long short-term memory (LSTM) networks
- Gated recurrent unit (GRU)
- Convolutional networks

These deep learning approaches offer flexibility and are valuable additions to your forecasting toolkit.

Understanding the Encoder-Decoder Paradigm

In Chapter 5, we discussed machine learning as learning a function that maps inputs to outputs: $y = h(x)$. For time series forecasting, this becomes:

$$Y_t = h(y_{t-1}, y_{t-2}, \ldots, y_{t-n})$$

where t is the current timestep, and n is the history available.

Deep learning, like other machine learning methods, learns a function that maps historical data to future values. In Chapter 11, we explored how deep learning uses representation learning to extract useful features. This concept is enhanced in time series by the encoder-decoder paradigm, which models variable-length inputs and outputs in an end-to-end manner.

While the origin of the encoder-decoder architecture is unclear, it gained prominence in 2014 through the work of Ilya Sutskever and Cho et al., who introduced the Seq2Seq and encoder-decoder models for machine translation.

Before diving deeper, let's briefly discuss feature/input spaces and latent spaces. The feature space is the vector space where the data resides, while the latent space is an abstract representation that encodes essential features of the data. For example, when recognizing a tiger, we don't remember every detail but rather a compressed understanding of its features, aiding quicker recognition. h for tasks like time series forecasting.

Encoder-Decoder Architecture

An encoder-decoder architecture consists of two main components:

- **Encoder:** Takes the input vector x and encodes it into a latent space, producing the latent vector z.
- **Decoder:** Takes the latent vector z and decodes it into the desired output $\hat{y}$.

The encoder-decoder architecture is visually represented as follows:

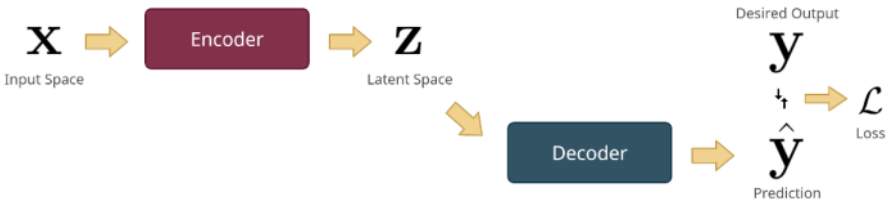


Figure 1: Encoder-Decoder Architecture

In time series forecasting, the encoder processes the history and retains the necessary information for the decoder to generate the forecast. Time series forecasting can be expressed as:

$$Y_t = h(y_{t-1}, y_{t-2}, \ldots, y_{t-n})$$

Using the encoder-decoder paradigm, this becomes:

$$Z_t = h(y_{t-1}, y_{t-2}, \ldots, y_{t-n})$$

where h is the encoder and g is the decoder, leading to the forecast:

$$Y_t = g(z_t)$$

The encoder and decoder can be customized with different architectures suited for time series forecasting.

Feed-Forward Networks (FFNs)

Feed-forward networks (FFNs) are the simplest type of neural network architecture. They consist of multiple perceptrons (linear units with non-linear activations) stacked together to form a network. In an FFN, information flows forward through the layers, which is why it is also called a fully connected network. Every unit in a layer is connected to every unit in the previous and next layers.

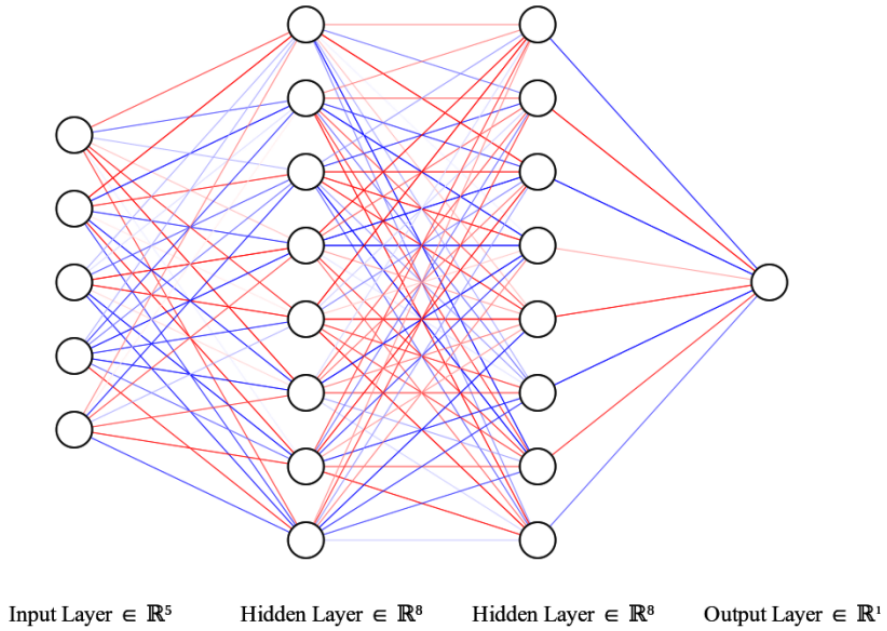


Figure 2: Encoder-Decoder Architecture

The network structure consists of:

- **Input Layer:** The first layer, equal to the input dimension.
- **Hidden Layers:** Layers between the input and output layers, defined by the number of units.
- **Output Layer:** The last layer, which is defined by the desired output (e.g., one unit for a single output, or multiple units for multiple outputs).

The two main hyperparameters are the number of hidden layers and the number of units in each hidden layer. For example, a network with two hidden layers and eight units per layer is shown in Figure 2

Feed-Forward Networks (FFN) in Time Series Forecasting

In time series forecasting, an FFN can serve both as an encoder and a decoder. As an encoder, we treat the time series problem as a regression task, similar to the approach in Chapter 5, Time Series Forecasting as Regression, by embedding time and feeding it into the FFN. As a decoder, the FFN takes the latent vector (the encoder's output) to produce the final forecast.

Recurrent Neural Networks (RNNs)

Recurrent neural networks (RNNs) are designed to handle sequential data and were first introduced by Rumelhart et al. (1986). RNNs address the limitations of FFNs in processing sequences by utilizing parameter sharing, where the same parameters are applied to different parts of the model. This allows RNNs to scale to longer sequences and recognize patterns across timesteps, overcoming FFN's inability to capture shifted motifs. For instance, an RNN can understand that "Tomorrow I will go to the bank"

and "I will go to the bank tomorrow" convey the same meaning, whereas an FFN cannot. Intuitively, RNNs apply the same FFN at each timestep with memory to store relevant information.

Recurrent Neural Networks (RNNs) - Sequence Processing

Consider a sequence with four elements: x_1 to x_4. An RNN block consumes an input and a hidden state (memory), producing an output. Initially, there is no memory, so we start with an initial hidden state H_0, typically an array of zeroes.

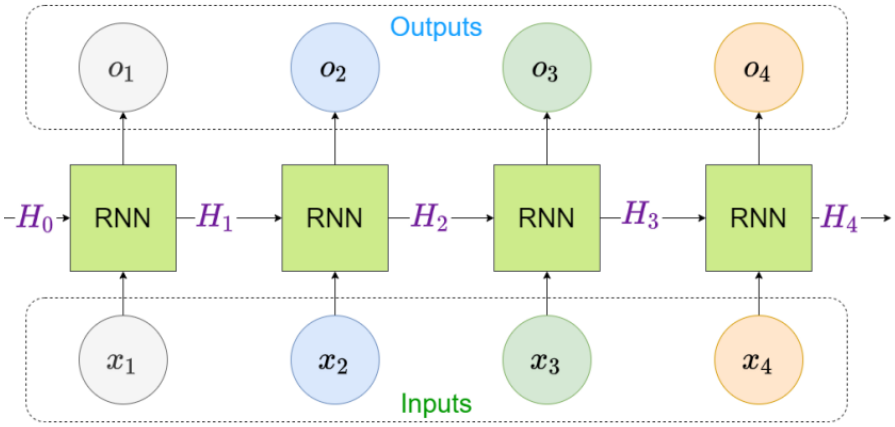


Figure 3: RNN Architecture

At each timestep, the RNN block takes the input x_t and the previous hidden state H_{t-1} to produce an output o_t and a new hidden state H_t. This process continues until the entire sequence is processed, resulting in outputs o_1 to o_4 and the final hidden state H_4. These outputs and the hidden state encode the information from the sequence and can be used for further tasks, such as prediction.

RNNs can be used in various configurations:

- **Many-to-One:** Many inputs lead to a single output, such as single-step forecasting or time series classification.
- **Many-to-Many:** Many inputs lead to many outputs, such as multi-step forecasting.

Recurrent Neural Networks (RNNs) - Equations and Concepts

Let the input to the RNN at time t be x_t, and the hidden state from the previous timestep be H_{t-1}. The updated equations are as follows:

$$A_t = W \cdot H_{t-1} + U \cdot x_t + b_1$$

$$H_t = \tanh(A_t)$$

$$o_t = V \cdot H_t + b_2$$

Here, U, V, and W are learnable weight matrices, and b_1 and b_2 are learnable bias vectors. The matrices

U, V, and W correspond to input-to-hidden, hidden-to-output, and hidden-to-hidden transformations, respectively. The tanh activation function helps decide what information to keep or forget, transforming the input into a latent dimension.

RNNs utilize a special form of backpropagation called *Backpropagation Through Time* (BPTT) to update weights. This process propagates gradients across timesteps, allowing RNNs to capture temporal dependencies. Deep learning frameworks like PyTorch provide ready-to-use RNN modules, making implementation easier.

Using RNNs in PyTorch

PyTorch provides ready-to-use RNN modules, which makes it easy to implement RNNs. You can simply import one of the RNN modules from the library and start using it in your code. However, before we start using these modules, it's important to understand a few more concepts.

Stacked RNNs

One important concept is stacking multiple layers of RNNs. In a stacked RNN, the outputs at each timestep from one RNN layer become the input to the RNN in the next layer. Each layer will have its own hidden state or memory, enabling hierarchical feature learning, which contributes to the success of deep learning.

Bidirectional RNNs

Another concept is bidirectional RNNs, introduced by Schuster and Paliwal in 1997. In a vanilla RNN, the inputs are processed sequentially from start to end. However, in a bidirectional RNN, there are two sets of input-to-hidden and hidden-to-hidden weights: one processes the inputs from start to end, and the other processes them in reverse (end to start). The hidden states from both directions are concatenated and used to generate the output.

The RNN Layer in PyTorch

Now, let's understand the PyTorch implementation for RNN. As with the Linear module, the RNN module is also available from `torch.nn`. Let's look at the different parameters the implementation provides while initializing:

- **input_size**: The number of expected features in the input. If we are using just the history of the time series, then this would be 1. However, when we use history along with other features, then this will be >1.
- **hidden_size**: The dimension of the hidden state. This defines the size of the input-to-hidden and hidden-to-hidden matrices.
- **num_layers**: The number of RNNs that will be stacked on top of each other. The default is 1.

- **nonlinearity**: The non-linearity to use. Although `tanh` is the originally proposed non-linearity, PyTorch also allows us to use `ReLU` (`relu`). The default is `'tanh'`.
- **bias**: This parameter decides whether or not to add bias to the update equations we discussed earlier. If the parameter is False, there will be no bias. The default is True.
- **batch_first**: There are two input data configurations that the RNN cell can use - we can have the input as (batch size, sequence length, number of features) or (sequence length, batch size, number of features). `batch_first = True` selects the former as the expected input dimensions. The default is False.
- **dropout**: This parameter, if non-zero, uses a dropout layer on the outputs of each RNN layer except the last. Dropout is a popular regularization technique where randomly selected neurons are ignored during training. The dropout probability will be equal to the `dropout` value. The default is 0.
- **bidirectional**: This parameter enables a bidirectional RNN. If True, a bidirectional RNN is used. The default is False.

To continue applying the model to the same synthetic data we generated earlier in this chapter, let's initialize the RNN model in the code section:

Inputs and Outputs of the RNN Cell

As opposed to the Linear layer we saw earlier, the RNN cell takes in two inputs: the input sequence and the hidden state vector. The input sequence can be either (batch size, sequence length, number of features) or (sequence length, batch size, number of features), depending on whether we have set `batch_first=True`. The hidden state is a tensor whose size is (D*number of layers, batch size, hidden size), where D = 1 for `bidirectional=False` and D = 2 for `bidirectional=True`. The hidden state is an optional input and will default to zero tensors if left blank.

There are two outputs of the RNN cell: an output and a hidden state. The output can be either (batch size, sequence length, D*hidden size) or (sequence length, batch size, D*hidden size), depending on `batch_first`. The hidden state has the dimension of (D*number of layers, batch size, hidden size). Here, D = 1 or 2 based on the `bidirectional` parameter.

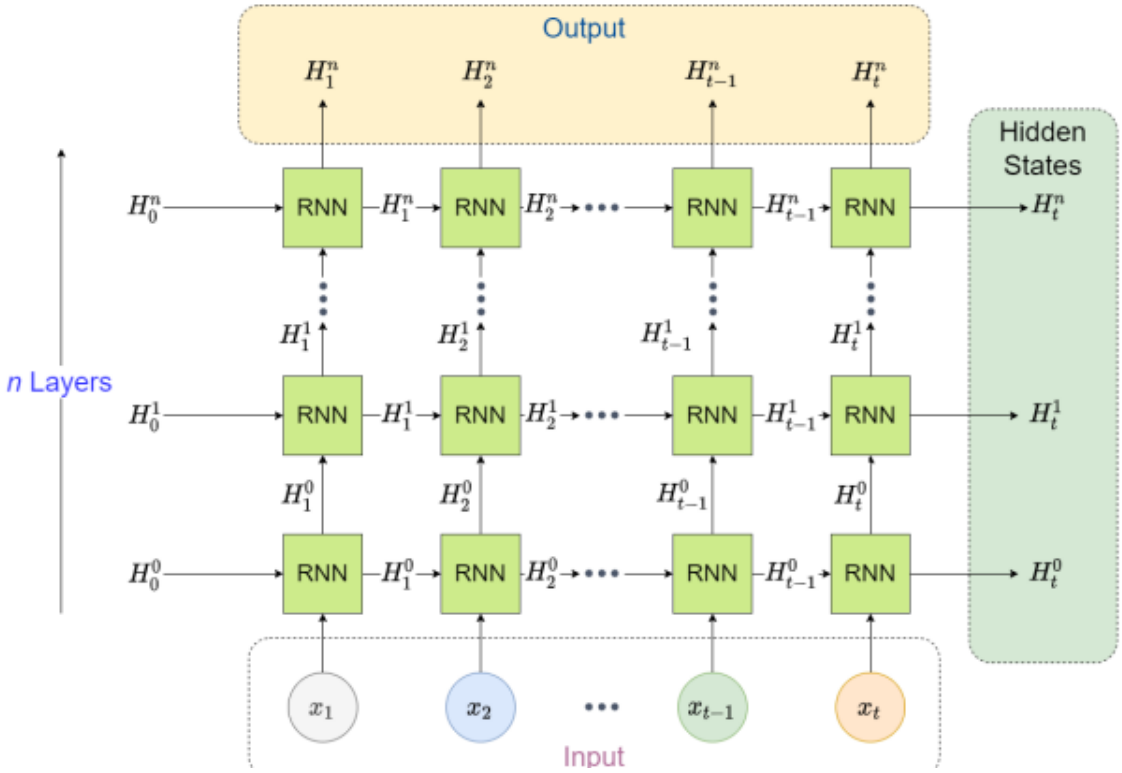
Example: Running a Sequence through an RNN

Let's run our sequence through an RNN and look at the inputs and outputs:

```
# input dim: torch.Size([6, 15, 1])
# batch size = 6, sequence length = 15 and number of features = 1, batch_first = True
output, hidden_states = rnn(rnn_input)


# output.shape -> torch.Size([6, 15, 32])
# hidden_states.shape -> torch.Size([1, 6, 32])
```

To make this clearer, let's look at it visually:



Pytorch implementation of stack RNN

Explanation of the Diagram

The hidden states at each timestep are used as input for the subsequent layer of RNNs, and the hidden states of the last layer of RNNs are collected as the output. But each layer has a hidden state (that's not shared with the others), and the PyTorch RNN collects the last hidden state from each layer and gives us that as well. Now, it is up to us to decide how to use these outputs. For instance, in a one-step-ahead forecast, we can use the output hidden states and stack a few linear layers on top of it to get the next timestep prediction. Alternatively, we can use the hidden states to transfer memory into another RNN as a decoder and generate predictions for multiple time steps. There are many more ways we can use this output, and PyTorch gives us that flexibility.

Challenges with RNNs

RNNs, while very effective in modeling sequences, have one big flaw. Because of BPTT, the number of units through which you need to backpropagate increases drastically with the length of the sequence to be used for training. When we have to backpropagate through such a long computational graph, we will encounter vanishing or exploding gradients. This is when the gradient, as it is backpropagated through the network, either shrinks to zero or explodes to a very high number. The former makes the network stop learning, while the latter makes the learning unstable.

Long short-term memory (LSTM) Networks

What are LSTMs?

Long Short-Term Memory (LSTM) networks are an advanced version of Recurrent Neural Networks (RNNs) introduced by Hochreiter and Schmidhuber in 1997. They solve key issues of vanishing and exploding gradients in vanilla RNNs. Inspired by logic gates in computers, LSTMs incorporate a special component called a **memory cell**, which acts as long-term memory, in addition to the hidden state used in classical RNNs.

Key Features of LSTMs

LSTMs use specialized mechanisms to manage and process information over long sequences. Here are the main features:

- **Memory Cell:** Acts like long-term memory, allowing the network to store important information over time.
- **Gates:** LSTMs use three types of gates to control the flow of information:
  - **Input Gate:** Decides how much new information from the input should be stored.
  - **Forget Gate:** Determines how much old information should be removed.
  - **Output Gate:** Selects what information from the memory cell should be output.

How LSTMs Work

At each time step, the LSTM processes:

- The current input, denoted as X_t.
- The hidden state from the previous time step, H_{t-1}.

The key components of LSTM operation are as follows:

- **Input Gate:**

$$I_t = \sigma(W_{xi} \cdot X_t + W_{hi} \cdot H_{t-1} + b_i)$$

- **Forget Gate:**

$$F_t = \sigma(W_{xf} \cdot X_t + W_{hf} \cdot H_{t-1} + b_f)$$

- **Output Gate:**

$$O_t = \sigma(W_{xo} \cdot X_t + W_{ho} \cdot H_{t-1} + b_o)$$

- **Cell State Update:**

$$\tilde{C}_t = \tanh(W_{xc} \cdot X_t + W_{hc} \cdot H_{t-1} + b_c)$$

$$C_t = F_t \odot C_{t-1} + I_t \odot \tilde{C}_t$$

- **Hidden State Update:**

$$H_t = O_t \odot \tanh(C_t)$$

Here, σ represents the sigmoid activation function, and $\odot$ denotes element-wise multiplication.

Implementation in PyTorch

PyTorch provides an easy-to-use implementation of LSTMs through the `nn.LSTM` module. The key difference from RNNs is that LSTMs use both:

- **Hidden State:** H_t, which captures short-term dependencies.
- **Cell State:** C_t, which captures long-term dependencies.

Below is an example of initializing and using an LSTM module in PyTorch:

```
import torch.nn as nn


lstm = nn.LSTM(
    input_size=1,      # Size of each input
    hidden_size=32,    # Size of the hidden state
    num_layers=5,      # Number of stacked LSTM layers
    batch_first=True   # Inputs are [batch, seq_length, features]
)


output, (hidden_states, cell_states) = lstm(input_data)
```

Why LSTMs are Important

LSTMs excel in handling sequential data like text, speech, and time-series data. Their ability to learn long-term dependencies makes them essential for tasks like:

- Language modeling and translation.
- Speech recognition.
- Stock price prediction and more.

article amsmath amssymb graphicx

Gated Recurrent Unit (GRU)

What is a GRU?

The Gated Recurrent Unit (GRU), introduced by Cho et al. in 2014, is a simpler alternative to LSTMs for handling sequential data. It uses fewer components and gates, making it less complex but still effective at managing long-term dependencies in data.

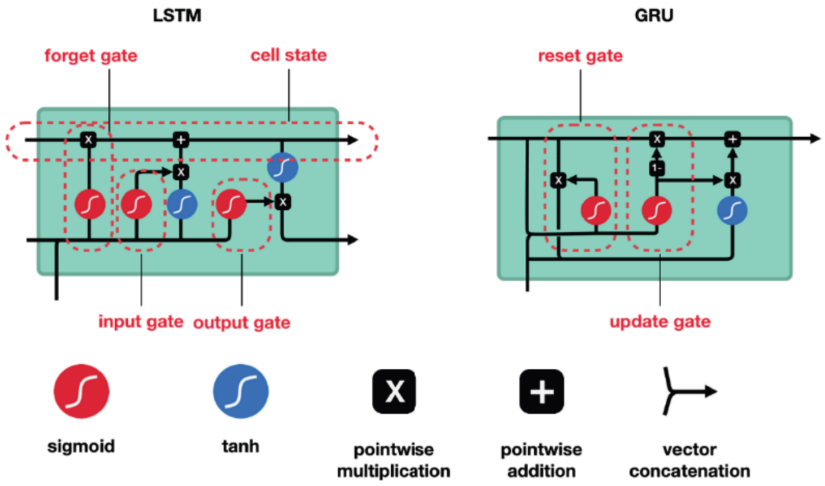


Figure 12.6 – A gating diagram of LSTM versus GRU

Key Features of GRU

- **Simpler Structure:** Unlike LSTMs, GRUs do not have a separate memory cell. Instead, the hidden state itself acts as the "gradient highway," carrying information through the network.
- **Two Gates Instead of Three:** GRUs use two gates to control information flow:
  - **Reset Gate:** Determines how much of the previous hidden state to include when calculating the current hidden state.

$$R_t = \sigma(W_{xr} \cdot X_t + W_{hr} \cdot H_{t-1} + b_r)$$

GRU in PyTorch

GRUs are implemented in PyTorch using the `nn.GRU` module, which works similarly to `nn.RNN` and `nn.LSTM`. The main difference lies in the internal update equations used by the GRU.

Example Initialization:

```
import torch.nn as nn

gru = nn.GRU(
    input_size=1,      # Input feature size
    hidden_size=32,    # Hidden state size
    num_layers=5,      # Number of stacked layers
    batch_first=True   # Inputs are in [batch, sequence, feature] format
)

output, hidden_states = gru(input_data)
```

```
print(output.shape)       # Output shape: [batch, sequence, hidden_size]
print(hidden_states.shape) # Hidden state shape: [layers, batch, hidden_size]
```

Why Use GRUs?

GRUs are simpler than LSTMs, requiring fewer parameters to train, which can make them faster and easier to implement. They are often preferred for tasks where model simplicity is important, while still being effective for sequential data like time-series analysis, speech recognition, or text generation.

Convolutional Networks (CNNs)

Convolutional Neural Networks (CNNs) are a type of neural network designed for processing data in grid-like structures such as images, time series, and 3D data from sensors. The key idea behind CNNs is inspired by human vision, where features like edges and patterns are detected from small regions of an image.

The concept of CNNs originated in 1979 with the Neocognitron, which was inspired by the human visual system. However, CNNs as we know them today were developed by Yann LeCun in 1989 using backpropagation. CNNs gained widespread popularity in 2012 when AlexNet won the ImageNet competition, showing the effectiveness of CNNs in image recognition.

At the core of CNNs is the **convolution** operation. Convolution involves applying a smaller matrix, called a kernel, to a larger grid (like an image) to extract useful features. The kernel slides over the input, performing element-wise multiplication with the input values and then summing them up to produce an output. This is repeated across the entire input.

1D Convolutions for Time Series

Although CNNs are most commonly used for images, they are also effective for sequences like time series. In 1D convolutions, the kernel slides over a sequence, much like how a moving window works in feature engineering (e.g., rolling means in time series). This allows CNNs to automatically learn patterns and features from the data.

Padding, Stride, and Dilations

- **Padding:** Adding extra values (usually zeros) to the input sequence to control the size of the output after convolution.
- **Stride:** The step size at which the kernel moves over the input. Larger strides reduce the output size.
- **Dilation:** Increases the spacing between the elements of the kernel, effectively increasing the receptive field without increasing the number of weights.

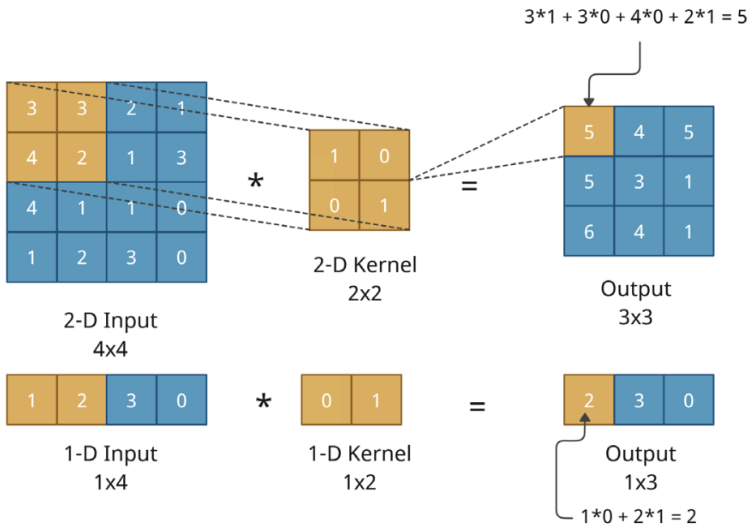


Figure 12.7 – A convolution operation on 2D and 1D inputs

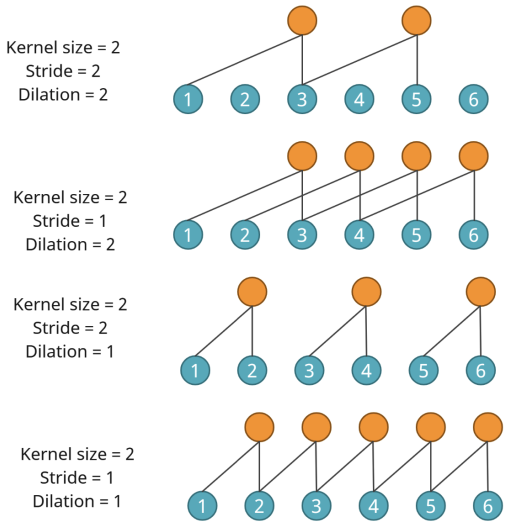


Figure 12.8 – Strides and dilations in convolutions

These parameters help control the size of the output and the receptive field of the CNN, which is crucial for capturing long-term dependencies in time series data.

Causal Convolutions

Causal convolutions ensure that predictions at any time step do not rely on future values, making them suitable for time series forecasting where future data is unavailable during training.

PyTorch CNN Implementation

In PyTorch, the CNN layer is implemented using `nn.Conv1d` for 1D data (like time series). The main parameters are:

- `in_channels`: Number of input features.

- `out_channels`: Number of filters applied.
- `kernel_size`: Size of the kernel.
- `stride, padding, and dilation`: Control the convolution behavior.
- `bias`: Adds a learnable bias term to the output.

By adjusting these parameters, CNNs can be used to learn complex patterns in time series and other sequential data.

In summary, CNNs are powerful tools for extracting features from grid-like data, and with modifications like padding, stride, and dilation, they can be adapted to work effectively with time series data.