## Chapter 4 – Dipping Toes in Deep Learning

In the previous chapter, you explored TensorFlow's model-building APIs and learned how to retrieve and manipulate data efficiently. Building on that foundation, this chapter introduces the practical use of **deep learning models** to solve real problems. The focus is on understanding how different types of deep neural networks are implemented and trained using TensorFlow and Keras.

Deep learning is an umbrella term that encompasses many algorithms, which can be categorized based on factors such as the type of data they process, their depth, and their architectural design. In this chapter, three major families of deep neural networks are introduced: **Fully Connected Networks (FCNs)**, **Convolutional Neural Networks (CNNs)**, and **Recurrent Neural Networks (RNNs)**. Mastering these architectures is an essential skill for practitioners, as they form the backbone of many state-of-the-art solutions across domains such as computer vision and time-series forecasting.

The chapter revisits FCNs and CNNs discussed earlier and expands on them with complete implementations. It also introduces RNNs, which are particularly suited for learning patterns in sequential or time-dependent data. By examining these models, the chapter provides a stepping stone toward understanding more advanced deep learning architectures.

---

### 4.1 Fully Connected Networks

To motivate the use of deep learning, the chapter introduces an image restoration problem inspired by damaged old photographs. The idea is to restore corrupted images using neural networks. As a simplified and well-understood starting point, handwritten digit images are used instead of real photographs. An **autoencoder**, which is a type of fully connected network, is selected as the initial model due to its suitability for reconstruction tasks.

The proposed autoencoder architecture consists of an input layer with 784 units corresponding to flattened image pixels, followed by three hidden layers with 64, 32, and 64 neurons respectively, all using ReLU activations. The output layer also has 784 units but uses a tanh activation function to reconstruct the original image. The choice of architecture is based on empirical intuition rather than formal hyperparameter optimization, which is acknowledged as an important but complex topic in deep learning.

---

#### 4.1.1 Understanding the Data

The MNIST handwritten digit dataset is used to train the autoencoder. This dataset contains grayscale images of digits ranging from 0 to 9, with each image representing a single digit. There

are ten distinct classes, making MNIST a standard benchmark for classification and reconstruction tasks.

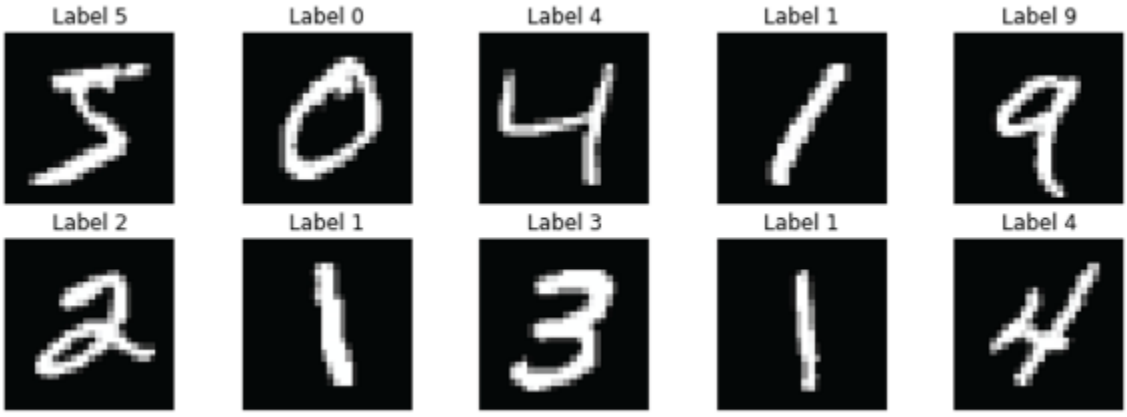



*Figure 4.1 Sample digit images. Each image contains a number from 0 to 9.*

In TensorFlow, the MNIST dataset can be loaded using a single function call, which returns training and testing splits. For this task, only the training images are used because the autoencoder performs unsupervised learning and does not require labels. Inspecting the training data reveals that it consists of 60,000 images, each with a shape of 28 × 28 pixels.

Although MNIST has historically played a central role in computer vision research, it is now considered too simple for evaluating modern models. As a result, alternative datasets such as Fashion-MNIST have been introduced, offering more challenging classification tasks while retaining the same data format.

Before feeding the data into the network, preprocessing steps are required. Pixel values are normalized from the range [0, 255] to [–1, 1] by subtracting 128 and dividing by 128. This normalization is crucial because the output layer uses a tanh activation function, which produces values within this range. Additionally, since fully connected networks expect one-dimensional inputs, each 28 × 28 image is reshaped into a vector of length 784.

To simulate damaged images, a corrupted version of the dataset is generated synthetically. This is achieved by randomly masking pixels using a binomial distribution, where each pixel has a 50% probability of being set to zero. The masking process introduces artificial noise, creating blacked-out regions in the images while preserving their overall structure.

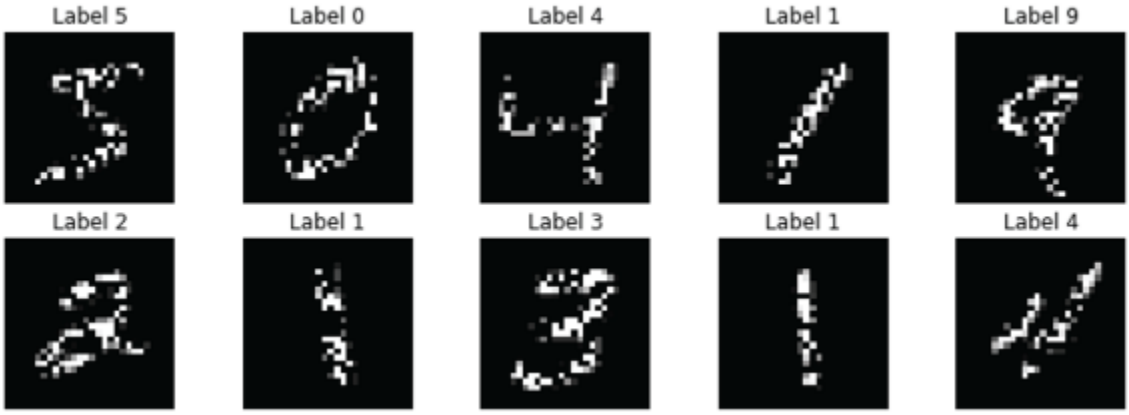



*Figure 4.2 Some of the synthetically corrupted images*

This corruption process produces input data that the autoencoder will attempt to reconstruct, enabling the model to learn how to restore missing or damaged pixel information.

### 4.1.2 Autoencoder Model

Autoencoders belong to the family of **Fully Connected Networks (FCNs)**, just like the multilayer perceptron (MLP) discussed earlier. They are categorized as FCNs because every neuron in one layer is fully connected to every neuron in the next layer. From a computational perspective, autoencoders and MLPs perform the same operations during the forward and backward passes. The key difference lies in their **training objective**. An MLP is trained in a supervised manner to predict labels, whereas an autoencoder is trained in an **unsupervised** manner to reconstruct its input, often from a corrupted or noisy version.

An autoencoder consists of two main functional phases: **compression** and **reconstruction**. In the compression phase, the input image—typically flattened into a vector—is mapped into a lower-dimensional hidden representation known as the **latent space**. In the reconstruction phase, this latent representation is used to reconstruct the original input as accurately as possible.

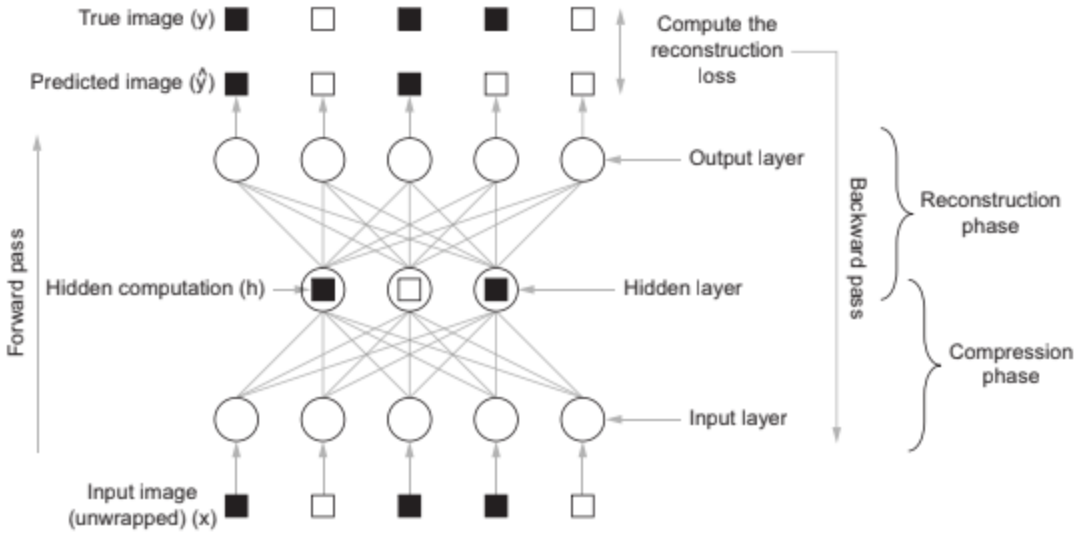



*Figure 4.3 A simple autoencoder with one layer for compression and another layer for reconstruction.*
*The black and white rectangles in the input image are the pixels present in the image.*

During the compression phase, the hidden representation is computed using a nonlinear transformation, typically ReLU, applied to a linear combination of inputs, weights, and biases. In mathematical terms, this corresponds to computing the hidden vector from the input using a weight matrix and bias. The reconstruction phase follows a similar computation, mapping the hidden representation back to the original input space. This sequence of transformations from input to output is known as the **forward pass**.

Once the output is generated, the model evaluates how close the reconstructed image is to the original image by computing a **reconstruction loss**, commonly the **mean squared error (MSE)**. The MSE measures the average squared difference between the original and reconstructed pixel values. This loss is computed over batches of images, and the model parameters are optimized to minimize it using gradient-based optimization. This optimization process constitutes the **backward pass**.

The autoencoder used in this section contains two compression layers and two reconstruction layers, implemented using the **Keras Sequential API**, which is well suited for simple, linear architectures. The model consists of four Dense layers. The first three layers use ReLU activation to progressively compress and then expand the latent representation, while the final layer uses a tanh activation function to ensure the output values lie in the range (–1, 1), matching the normalized input data.

The model is compiled using the mean squared error loss and the Adam optimizer. Once compiled, it is trained using corrupted images as inputs and the original images as targets. Training is performed over multiple epochs with mini-batches, and the steadily decreasing loss

indicates that the model is learning to reconstruct the images effectively. Because this is an unsupervised task, traditional metrics such as accuracy are not applicable.

This specific model is an example of a **denoising autoencoder**, a variant designed to recover clean inputs from noisy or partially corrupted data. By learning to remove noise, the autoencoder captures meaningful structure in the data rather than simply memorizing pixel values.

After training, the model is evaluated by feeding it newly corrupted images generated using a different random mask than the one seen during training. The reconstructed outputs demonstrate that the model can generalize its denoising capability beyond the exact noise patterns encountered during training.

*Figure 4.4 Images restored by the model. It seems our model is doing a good job.*

Beyond image restoration, autoencoders are valuable for learning **unsupervised feature representations** from large amounts of unlabeled data. These learned representations can be reused for downstream tasks such as image classification, often reducing the need for large labeled datasets. Autoencoders can also serve as dimensionality reduction tools, where the latent representation acts as a compact proxy for clustering or visualization.

In summary, this section introduced the autoencoder as a fully connected, unsupervised learning model capable of reconstructing damaged images. You examined its architecture, implemented it using the Keras Sequential API, trained it on the MNIST dataset, and evaluated its performance through reconstruction quality and loss reduction. This establishes a strong foundation for understanding more advanced deep learning models, leading naturally into the discussion of convolutional neural networks in the next section.

### 4.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a class of deep learning models specifically designed for **computer vision tasks**. In this scenario, CNNs are explored in the context of a real-world application: predicting the presence of vehicles in road images as part of a traffic congestion modeling system. Before investing time and resources into collecting and labeling custom traffic data, a practical approach is to evaluate feasibility using a well-known benchmark data set. For this purpose, the **CIFAR-10** data set is selected.

CIFAR-10 is well suited for this experiment because it contains multiple vehicle-related classes (such as automobiles and trucks) alongside non-vehicle objects (such as animals). Achieving good classification performance on this data set would indicate that CNN-based approaches are promising for the larger traffic modeling problem. Since CNNs are known to excel at extracting spatial features from images, they are a natural choice for this task.

---

### 4.2.1 Understanding the Data

The CIFAR-10 data set consists of **60,000 RGB images**, each with a resolution of **32 × 32 pixels**. The data is split into **50,000 training samples** and **10,000 testing samples**, covering **10 distinct object classes**. These classes include vehicles and animals, making the data set diverse enough to evaluate object classification performance.


*Figure 4.5 Sample images from cifar-10 data set along with their labels*

The data set is loaded using the tensorflow-datasets package, which returns a dictionary containing separate train and test splits. Each split is represented as a tf.data.Dataset, where each data point includes an image, its corresponding label, and a unique identifier. From inspection, two important characteristics of the raw data become apparent. First, images are stored as **unsigned 8-bit integers (uint8)**, and second, labels are provided as **integer class indices**, not one-hot encoded vectors.

Because neural network model parameters typically use floating-point precision, the images must be converted to float32. Additionally, since the classification task involves multiple classes, labels are converted into **one-hot encoded vectors**. This preprocessing step ensures that both inputs and outputs are compatible with the CNN model and its loss function.

A simple formatting function is defined to perform these transformations. This function casts image tensors to float32 and converts integer labels into one-hot encoded vectors with a depth

equal to the number of classes. The function is applied to every element in the training data set using the map() method, after which the data is grouped into mini-batches of size 32.

After batching, each element produced by the data pipeline consists of a tuple containing a batch of images with shape **(32, 32, 32, 3)** and a batch of labels with shape **(32, 10)**. This confirms that the data is now in the correct format to be consumed by a convolutional neural network.

At this point, the CIFAR-10 data has been successfully loaded, transformed, and batched. The images and labels are now fully prepared for training a CNN model, which will be discussed in the subsequent sections.

### 4.2.2 Implementing the Network

To classify images from the CIFAR-10 data set, a **Convolutional Neural Network (CNN)** is employed. CNNs have become the dominant architecture for computer vision tasks due to two fundamental advantages. First, CNNs process images while preserving their **spatial structure**, keeping the height and width dimensions intact, whereas fully connected layers require flattening images into one-dimensional vectors, resulting in a loss of locality information. Second, CNNs are **highly parameter-efficient**, because convolution operations apply small kernels across the entire image using shared weights, unlike fully connected layers where every input node connects to every output node.

A CNN architecture consists of **interleaved convolution and pooling layers**, followed by one or more **fully connected layers**. Consequently, CNNs are built using three main types of layers: convolution layers, pooling layers, and fully connected layers. Each of these layers plays a distinct role in extracting features and producing final predictions.

A **convolution layer** is composed of multiple filters (also known as convolution kernels) that slide over the input image to produce feature maps. Each feature map represents how strongly a particular filter is activated at different spatial locations in the image. For example, if a filter represents a vertical edge, its feature map highlights where vertical edges appear and how strongly they are present. In more complex tasks such as face recognition, filters may learn patterns such as the shape of an eye, activating strongly in regions where that feature appears.

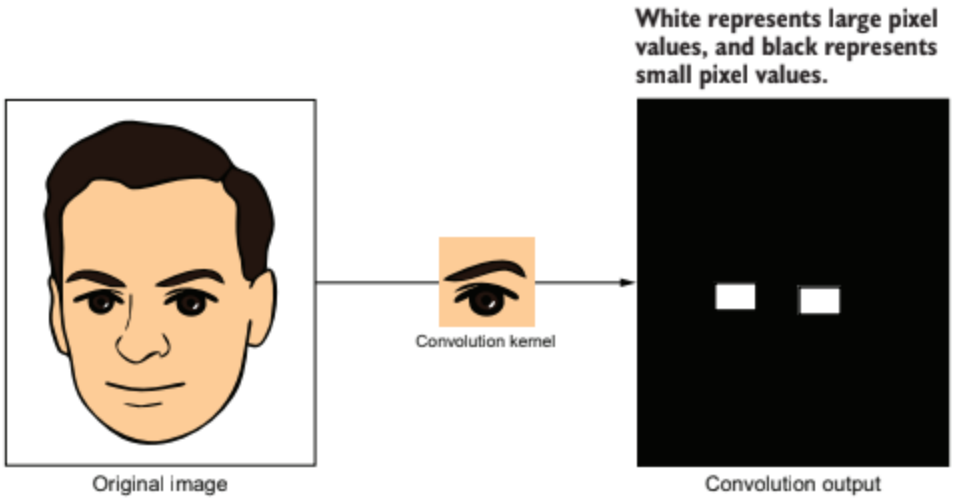



*Figure 4.6 The result of a convolution operation at a very abstract level*

An important property of convolutional networks is that **deeper layers learn increasingly abstract features**. Lower layers tend to detect simple features such as edges or lines, while higher layers capture more complex structures, such as facial components and their spatial relationships.

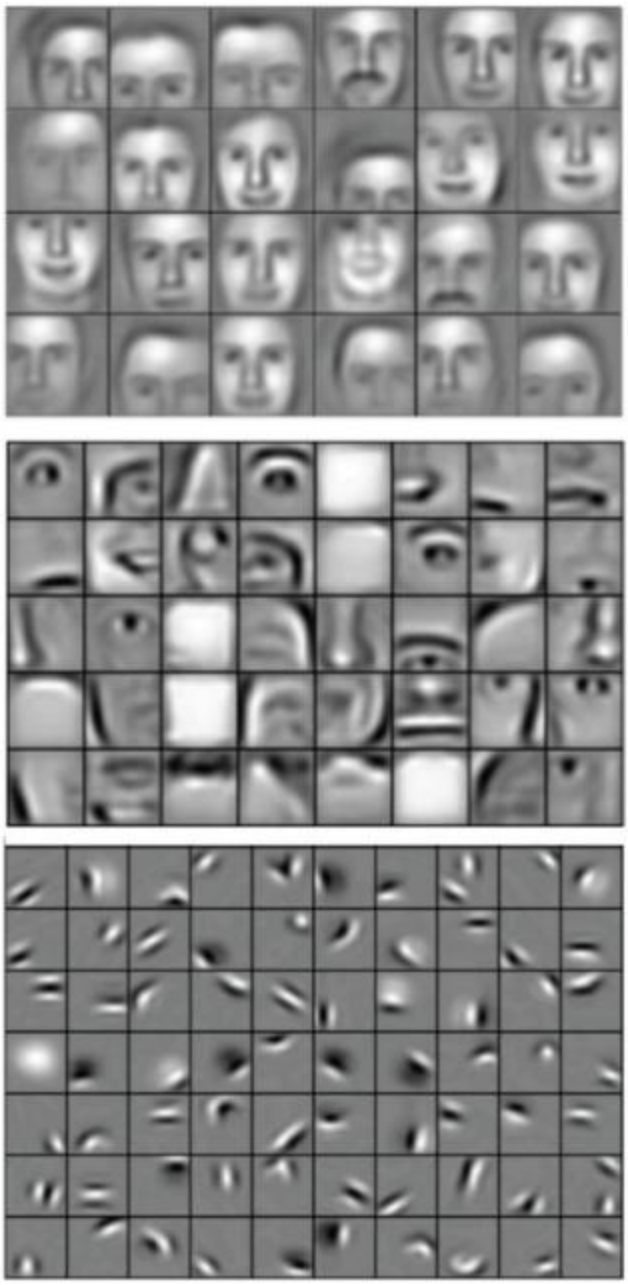


*Figure 4.7 Features learned by a convolutional neural network*

Following convolution layers, **pooling layers** are applied to reduce the spatial dimensions (height and width) of the feature maps. This reduction helps the model become **translation invariant**, meaning that objects can still be recognized even if their position shifts slightly relative to what was seen during training. Pooling therefore improves robustness while also reducing computational complexity.

After convolution and pooling operations, the network must eventually produce a probability distribution over classes. However, convolution and pooling layers output **three-dimensional**

**tensors** consisting of height, width, and channel dimensions, while fully connected layers require **one-dimensional inputs**. To resolve this mismatch, the multidimensional output is **flattened** into a single vector, analogous to unwrapping an RGB image into a long one-dimensional array. The final fully connected layer applies a **softmax activation** to transform the raw scores into a valid probability distribution.

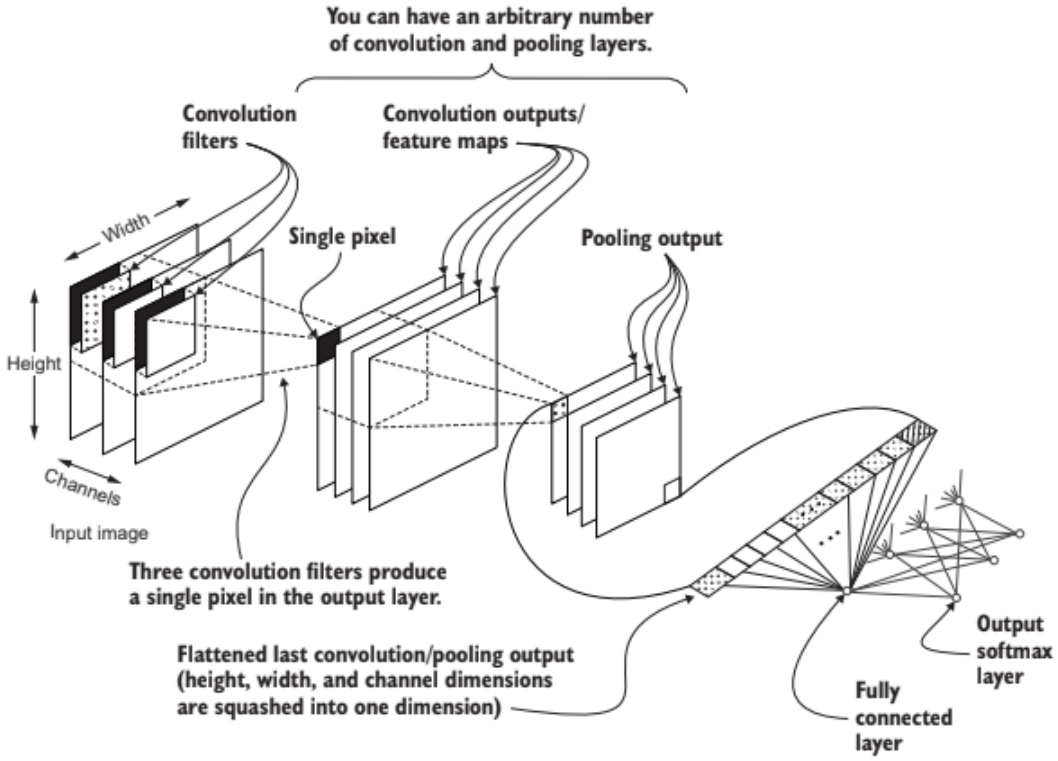



*Figure 4.8 A simple CNN architecture*

The CNN described in this section is implemented using the **Keras Sequential API** and consists of three convolution layers followed by two fully connected layers.

**Listing 4.2 Defining a CNN with the Keras Sequential API**

```
from tensorflow.keras import layers, models
import tensorflow.keras.backend as K

K.clear_session()
```

Clearing any existing Keras states (e.g., models) to start fresh



```
cnn = models.Sequential(
    [layers.Conv2D(
        filters=16, kernel_size= (9,9), strides=(2,2), activation='relu',
           padding='valid', input_shape=(32,32,3)
      ),
    layers.Conv2D(
        filters=32, kernel_size= (7,7), activation='relu', padding='valid'
    ),
    layers.Conv2D(
        filters=64, kernel_size= (7,7), activation='relu', padding='valid'
    ),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')]
)
```

Defining a convolution layer; it takes parameters like filters, kernel_size, strides, activation, and padding.

Creating an intermediate fully connected layer

Final prediction layer

Before feeding the data to a fully connected layer, we need to flatten the output of the last convolution layer.

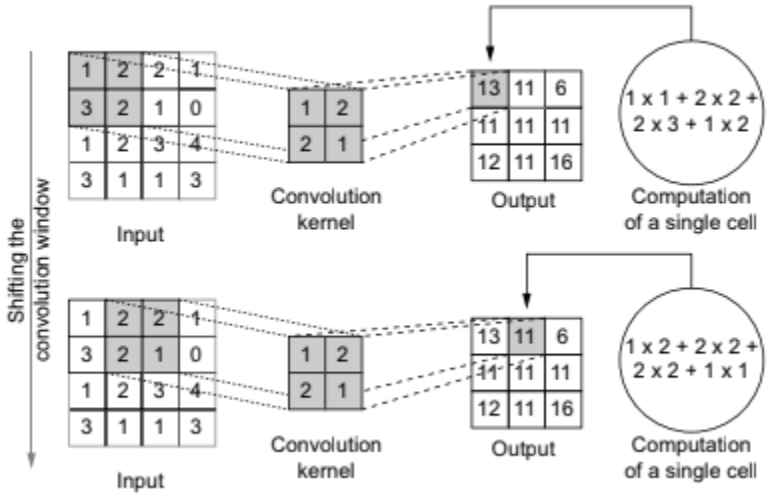
*Listing 4.2 Defining a CNN with the Keras Sequential API*

The first convolution layer is defined using layers.Conv2D, which is Keras's abstraction of the two-dimensional convolution operation. Internally, it performs the same computation as tf.nn.convolution, while hiding many implementation details. Several parameters must be specified for this layer, including the number of filters, kernel size, stride, activation function, padding, and input shape.

The parameters such as the number of filters, kernel size, stride, activation function, and the number of units in dense layers are known as **hyperparameters**. In principle, these should be selected through hyperparameter optimization techniques. However, in this implementation, the hyperparameters are chosen empirically.

The **stride** determines how many pixels the convolution window skips while sliding across the image. Larger strides result in faster spatial reduction. The **padding** parameter controls how the convolution handles image borders. Without padding (valid padding), output dimensions shrink automatically, whereas with same padding, artificial borders are added to preserve the input size.

The convolution operation itself consists of sliding a kernel across the input image and computing the sum of element-wise products at each position.




*/Figure 4.9 The computations that happen in the convolution operation while shifting the window*

Several hyperparameters influence the output size and behavior of a convolution layer: the number of filters, kernel height and width, stride, and padding. A convolution layer typically contains multiple filters, each learning a different feature. When an image tensor with height, width, and channels is convolved with *n* filters, the output is a tensor with the same spatial dimensions (subject to stride and padding) and *n* channels.

CNNs operate on **batches of data**, which means that both the input and output of a Conv2D layer are **four-dimensional tensors** with dimensions corresponding to batch size, height, width, and channels. The filters themselves are also four-dimensional tensors defined by kernel height, kernel width, incoming channels, and outgoing channels.

| | Dimensionality | Example |
|---|---|---|
| Input | [batch size, height, width, in channels] | [32, 64, 64, 3] (i.e., a batch of 32, 64 × 64 RGB images) |
| Convolution filters | [height, width, in channels, out channels] | [5, 5, 3, 16] (i.e., 16 convolution filters of size 5 × 5 with 3 incoming channels) |
| Output | [batch size, height, width, out channels] | [32, 64, 64, 16] (i.e., a batch of 32, 64 × 64 × 16 tensors) |

*Table 4.1 The dimensionality of the input, filters, and the output of a Conv2D layer*

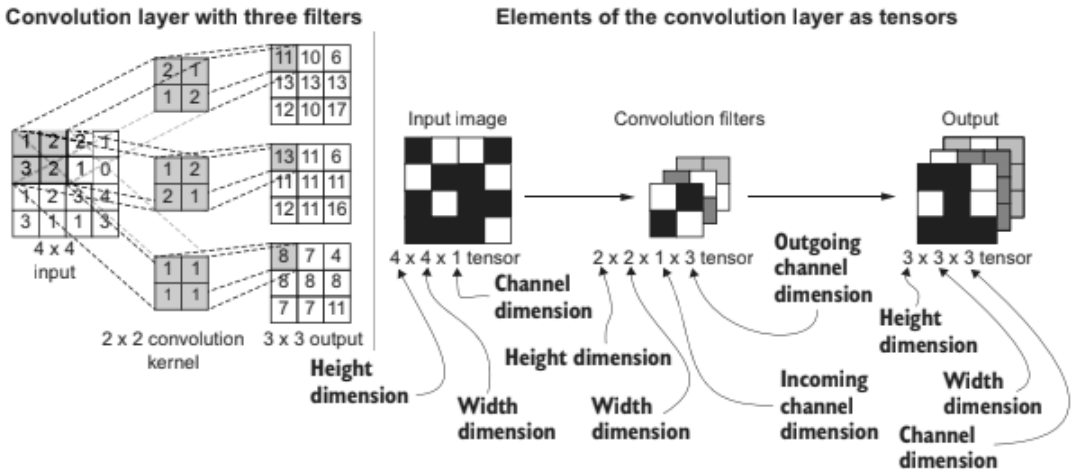



*Figure 4.10 A computation of a convolution layer with multiple filters*

Kernel size directly affects the output dimensions of a convolution. Increasing the kernel size reduces the output size and increases the number of parameters. The output size can be computed using the relationship

size(y) = size(x) − size(f) + 1.

For this reason, CNNs typically favor **smaller kernels**, which encourage learning robust features with fewer parameters and better generalization.

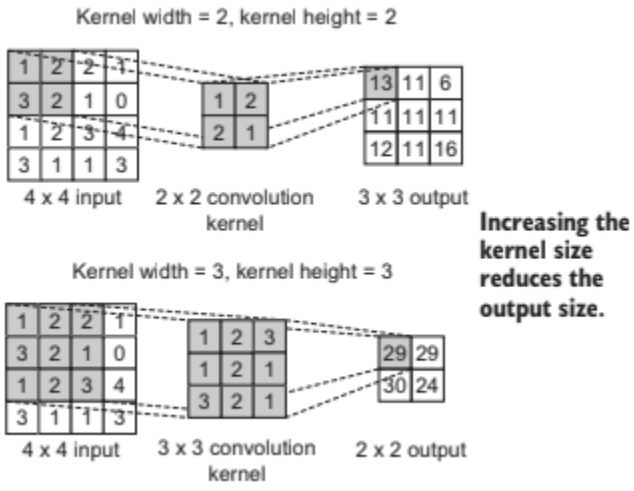



*Figure 4.11 Convolution operation with a kernel size of 2 and kernel size of 3.Increasing the kernel size leads to a reduced output size.*

The **stride** parameter controls how much the convolution window shifts at each step. Increasing the stride further reduces the output size without changing the kernel size, allowing explicit control over spatial reduction.

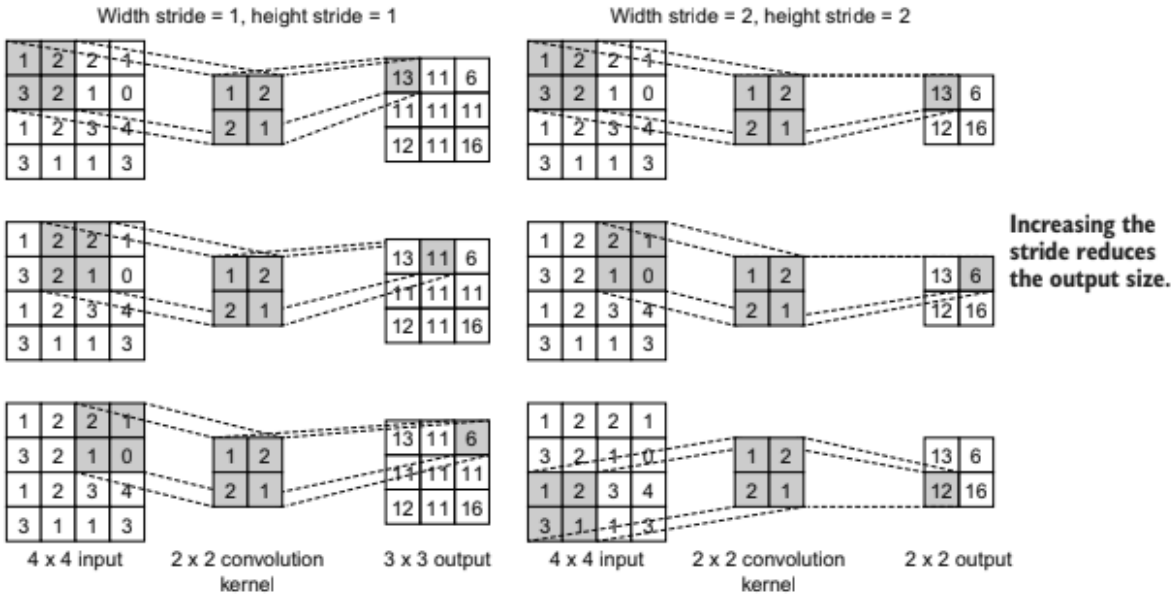



*Figure 4.12 Convolution operation with stride = 1 versus stride = 2*

Padding addresses a critical limitation of deep CNNs. Without padding, repeated convolution operations shrink the spatial dimensions rapidly, eventually producing invalid or zero-sized outputs. **Same padding** alleviates this issue by adding a border around the image, preserving spatial dimensions across layers. In contrast, **valid padding** performs no padding and leads to dimensional reduction.

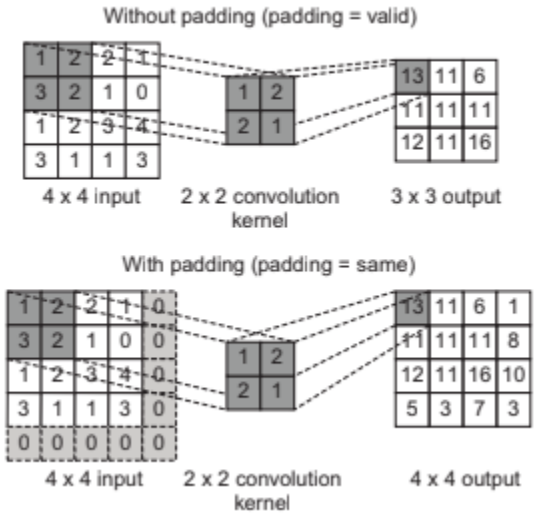
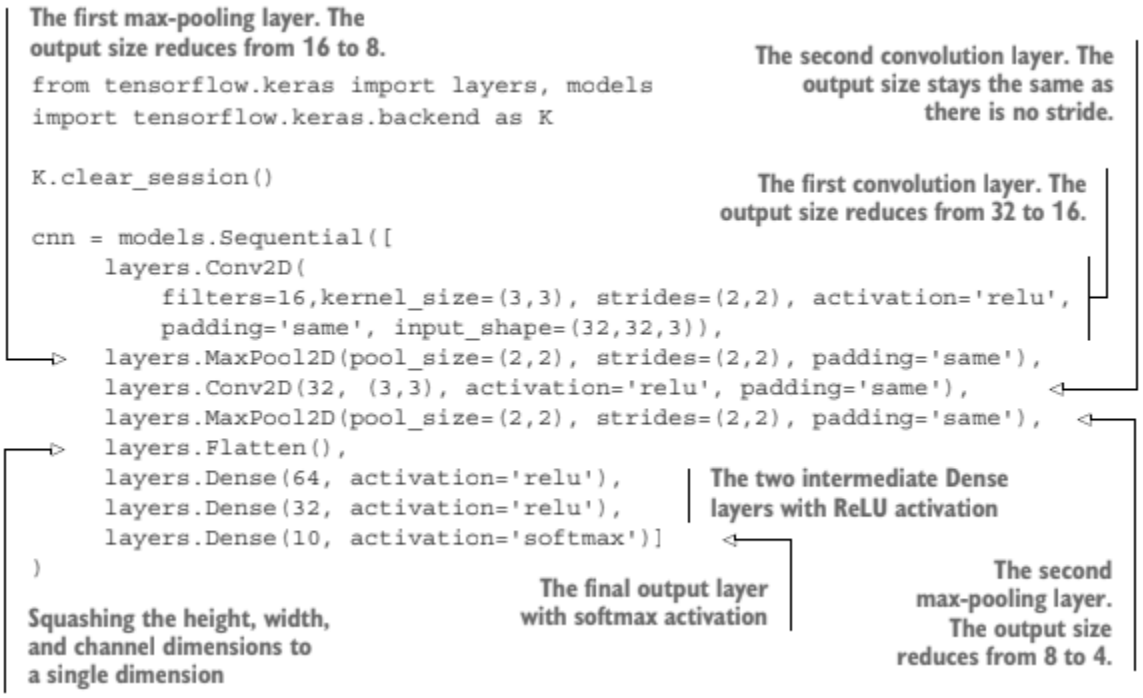


*Figure 4.13 Valid versus same padding*

When implementing the initial CNN, an error occurs due to invalid output dimensions caused by aggressive kernel sizes and lack of padding. By computing the output sizes layer by layer, it becomes evident that the final convolution produces a zero-sized output. This issue is resolved by introducing **same padding** and **max-pooling layers**, which allow controlled dimensional reduction while preventing negative dimensions.




```
from tensorflow.keras import layers, models
import tensorflow.keras.backend as K

K.clear_session()

cnn = models.Sequential([
      layers.Conv2D(
          filters=16,kernel_size=(3,3), strides=(2,2), activation='relu',
          padding='same', input_shape=(32,32,3)),
      layers.MaxPool2D(pool_size=(2,2), strides=(2,2), padding='same'),
      layers.Conv2D(32, (3,3), activation='relu', padding='same'),
      layers.MaxPool2D(pool_size=(2,2), strides=(2,2), padding='same'),
      layers.Flatten(),
      layers.Dense(64, activation='relu'),
      layers.Dense(32, activation='relu'),
      layers.Dense(10, activation='softmax')]
)
```

*Listing 4.3 The corrected CNN model that has positive dimensions*

Max-pooling layers reduce spatial dimensions by selecting maximum values within a sliding window, and their hyperparameters closely mirror those of convolution layers. In the corrected model, padding is set to 'same' throughout, and strides are used intentionally to manage spatial reduction. After the convolution and pooling layers, the output is flattened and passed through fully connected layers with ReLU activation, followed by a final softmax layer for classification.

The model is compiled using categorical cross-entropy loss, the Adam optimizer, and accuracy as the evaluation metric. Training the CNN on CIFAR-10 data shows steady loss reduction and achieves training accuracy above 70%, indicating strong learning performance.

Finally, it is noted that the **first fully connected layer after convolution layers often becomes a performance bottleneck**, as it may contain a disproportionately large number of parameters. Careful design of this layer is essential to avoid memory issues and inefficiencies.

This section demonstrates how CNNs can be effectively implemented for image classification, highlights the importance of understanding convolutional hyperparameters, and shows how architectural choices directly impact model correctness and performance.

### 4.3 One Step at a Time: Recurrent Neural Networks (RNNs)

In this section, the focus shifts from image-based learning to **time-series modeling**, using atmospheric $CO_2$ concentration data as a practical example. The task is to predict future $CO_2$ concentration levels based on historical measurements collected over the past three decades. Unlike previous problems, where each input sample was treated as independent, this problem exhibits **temporal dependency**, meaning that the current value depends on values observed in previous time steps.

Traditional feed-forward neural networks, such as fully connected networks and CNNs, assume that inputs are **independent and identically distributed (i.i.d.)**. This assumption does not hold for time-series data, where sequential order and historical context are essential. To address this limitation, a specialized class of neural networks known as **Recurrent Neural Networks (RNNs)** is introduced. RNNs differ from feed-forward networks by incorporating an internal memory that allows information from previous time steps to influence predictions at the current time step.

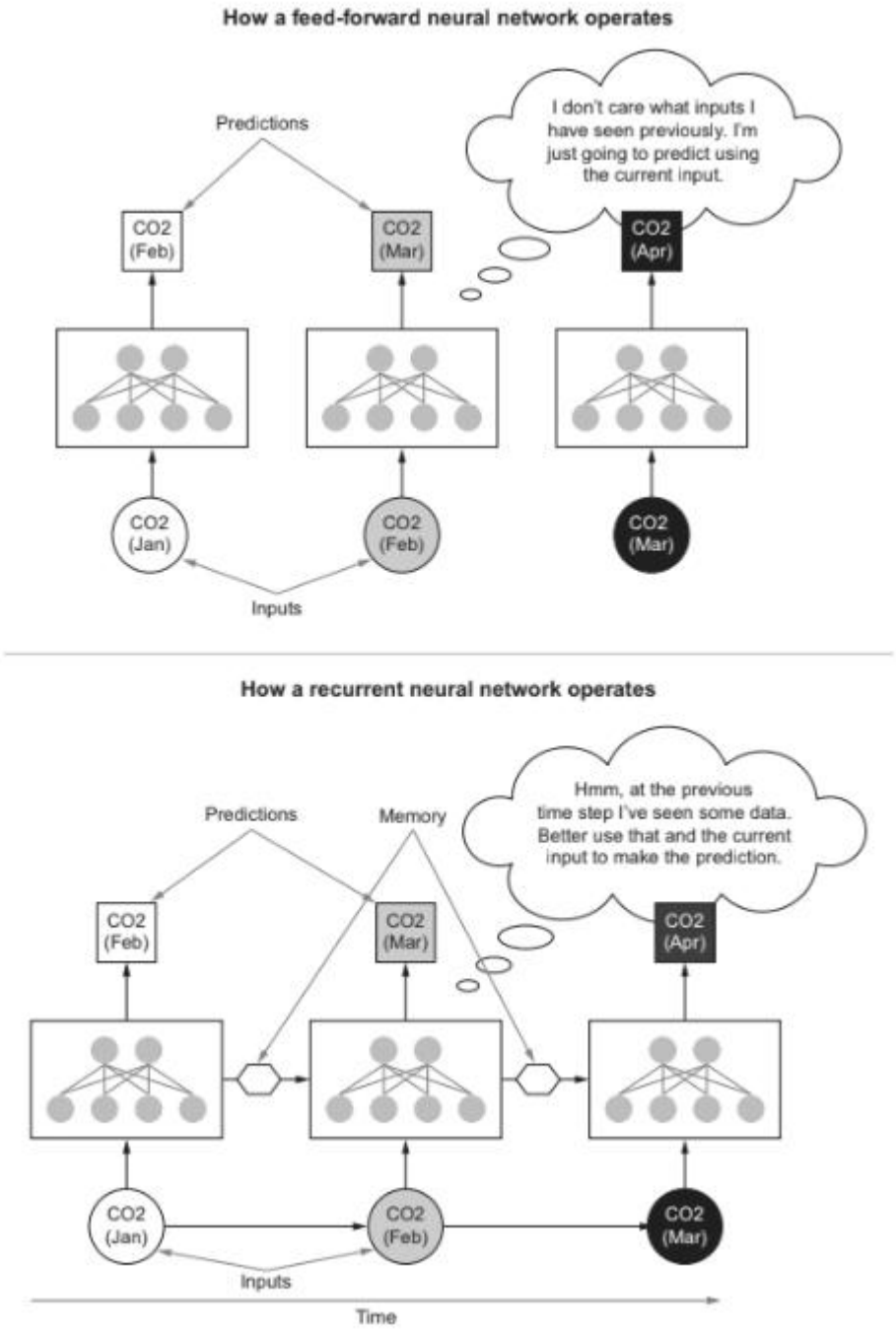



*Figure 4.14 The operational difference between a feed-forward network and an RNN*

### 4.3.1 Understanding the Data

The data set used in this section consists of monthly global $CO_2$ concentration measurements provided in CSV format. Each data point contains a date and a floating-point value representing $CO_2$ concentration. The data is downloaded programmatically and loaded into a pandas DataFrame for processing.

Once loaded, the **Date** column is set as the index of the DataFrame to facilitate time-based visualization. A line plot of the **Average** $CO_2$ concentration reveals two dominant characteristics: a strong long-term upward trend and short-term periodic fluctuations.

| | Date | Decimal Date | Average | Trend |
|---|---|---|---|---|
| 0 | 1980-01-01 | 1980.042 | 338.45 | 337.82 |
| 1 | 1980-02-01 | 1980.125 | 339.14 | 338.10 |
| 2 | 1980-03-01 | 1980.208 | 339.46 | 338.12 |
| 3 | 1980-04-01 | 1980.292 | 339.86 | 338.24 |
| 4 | 1980-05-01 | 1980.375 | 340.30 | 338.77 |

*Figure 4.15 Sample data in the data set*

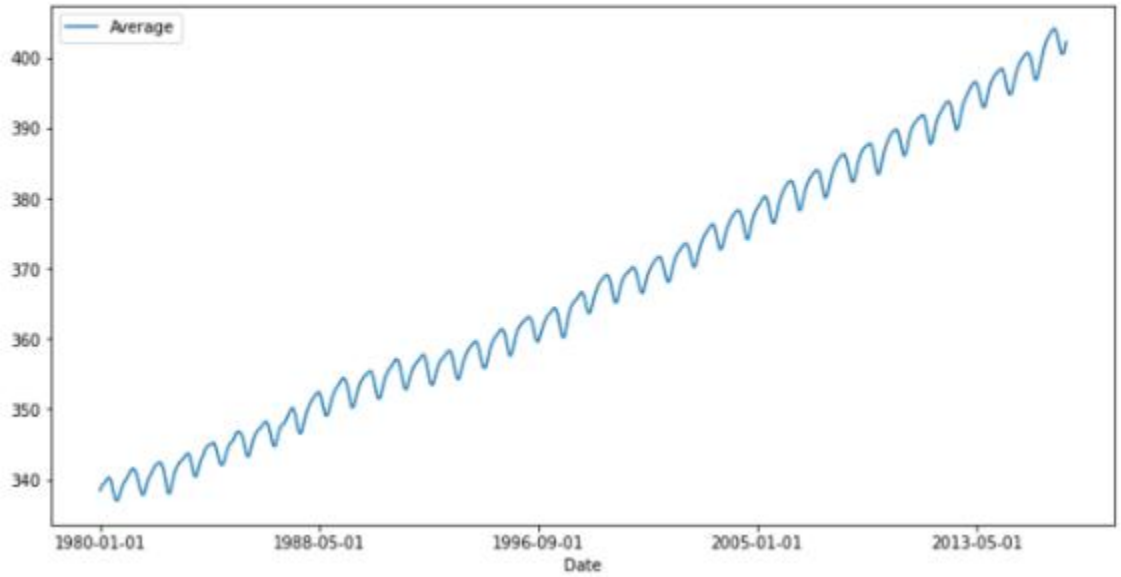



*Figure 4.16 $CO_2$ concentration plotted over time*

The pronounced upward trend presents a modeling challenge. Because the data range continuously increases over time, a model trained on earlier values may struggle to generalize to future values that lie outside the observed range. Feeding such non-stationary data directly into a neural network typically leads to poor performance.

To mitigate this issue, the data is transformed to focus on **relative changes** rather than absolute values. A new column, **Average Diff**, is created by subtracting each value from its previous time step. This transformation converts the original increasing sequence into a series that fluctuates within a small, stable range.

| Date | Decimal data | Average | Trend | Average diff |
|---|---|---|---|---|
| 1980-01-01 | 1980.042 | 338.45 | 337.83 | 0.00 |
| 1980-02-01 | 1980.125 | 339.15 | 338.10 | 0.70 |
| 1980-031-01 | 1980.208 | 339.48 | 338.13 | 0.33 |
| 1980-04-01 | 1980.292 | 339.87 | 338.25 | 0.39 |
| 1980-05-01 | 1980.375 | 340.30 | 338.78 | 0.43 |

*Table 4.2 Sample data after introducing the Average Diff column*

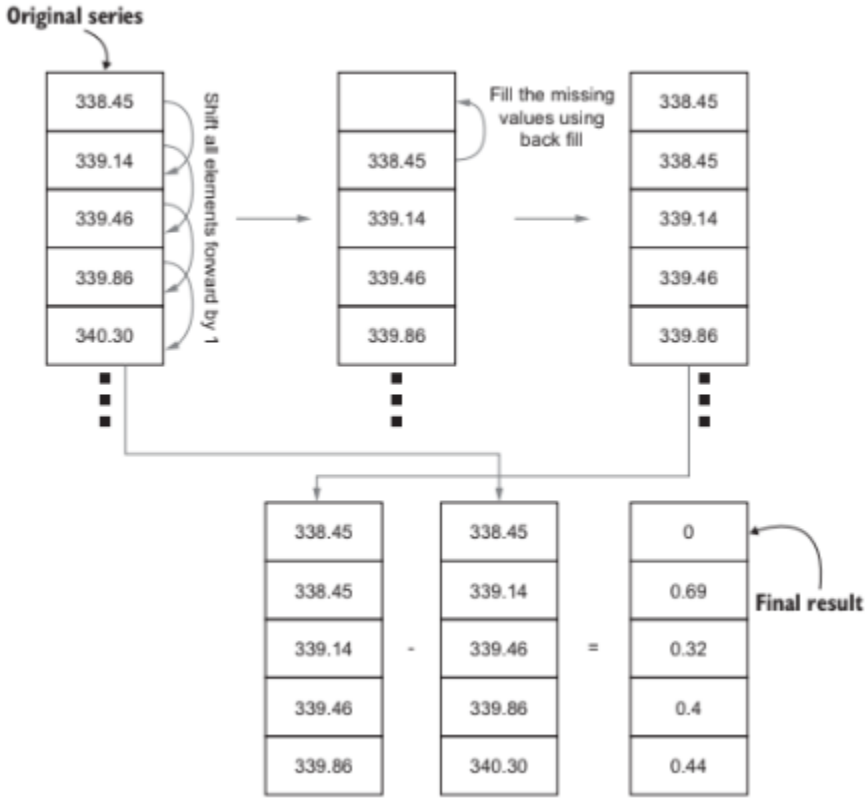



*Figure 4.17 Transformation from Average to Average Diff*

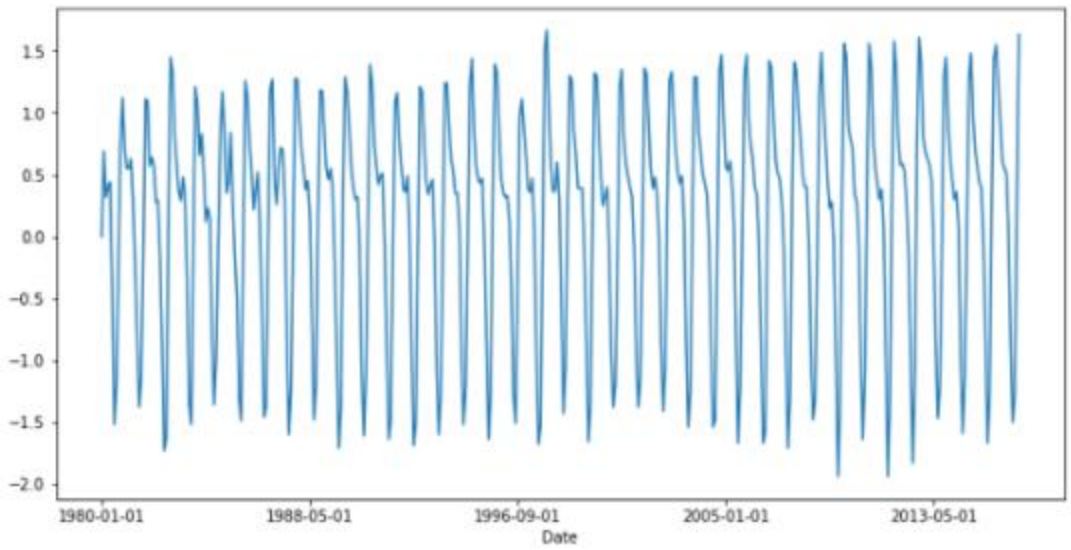



*Figure 4.18 Relative change of $CO_2$ concentration over time*

This transformation significantly improves the suitability of the data for sequence modeling, as the resulting values remain within a narrow range and better capture short-term dynamics.

**Creating Time-Series Batches**

Preparing training data for time-series models differs fundamentally from standard supervised learning. Randomly sampling individual data points would break the temporal structure of the sequence. Instead, training samples must preserve the **order of observations**.

In this example, the model uses the previous **12 time steps** to predict the next value in the sequence. The number of time steps is a **hyperparameter** that must be selected carefully, balancing the temporal patterns present in the data with the memory capacity of the model. Each training sample therefore consists of a sequence of 12 consecutive values as input and the 13th value as the target.

By randomly selecting starting positions in the full time series and extracting consecutive subsequences, batches can be constructed that preserve temporal dependencies while still introducing randomness at the batch level. This approach allows efficient training while respecting the sequential nature of the data.

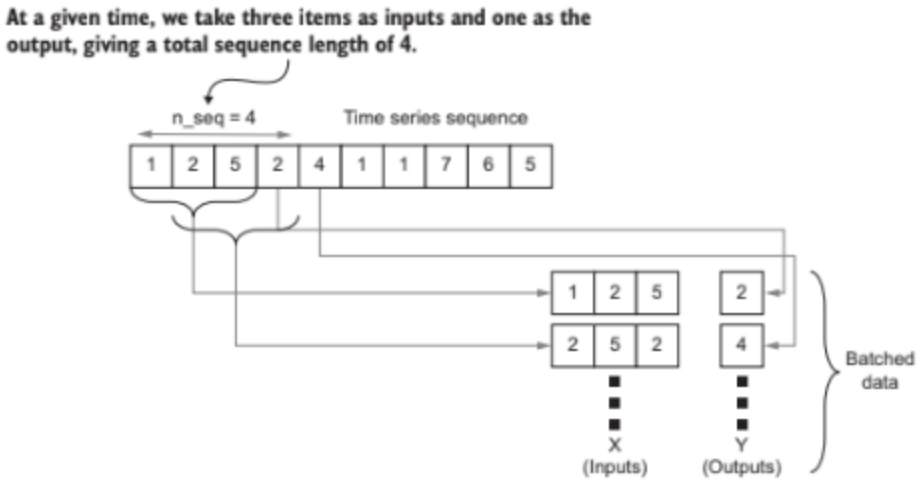



*Figure 4.19 Batching time-series data*

To automate this process, a Python function is implemented that generates **all possible consecutive sequences** of length 12 as inputs and their corresponding next values as outputs. These sequences can then be shuffled and fed into the model during training.

```
import numpy as np

def generate_data(co2_arr,n_seq):
    x, y = [],[]
    for i in range(co2_arr.shape[0]-n_seq):
        x.append(co2_arr[i:i+n_seq-1])
        y.append(co2_arr[i+n_seq-1:i+n_seq])
    x = np.array(x)
    y = np.array(y)
    return x,y
```

Extracting a sequence of values n_seq long

Extracting the next value in the sequence as the output

Combining everything into an array

*Listing 4.4 Code for generating time-series data sequences*

### 4.3.2 Implementing the Model

With a solid understanding of the time-series data, the next step is to implement a neural network capable of learning temporal dependencies. The model architecture is deliberately kept simple to demonstrate the core concepts behind recurrent neural networks. It consists of a **SimpleRNN layer with 64 hidden units**, followed by a **Dense layer with 64 units and a ReLU activation**, and a final **Dense layer with a single linear output** to support regression.

The SimpleRNN layer is the most critical component of the network, as it enables learning from sequential data by maintaining an internal memory that evolves over time. The remaining Dense layers transform the learned temporal representation into a single continuous output representing the predicted $CO_2$ value.

from tensorflow.keras import layers, models

rnn = models.Sequential([

```
    layers.SimpleRNN(64),
    layers.Dense(64, activation='relu'),
    layers.Dense(1)
])
```

The design choices, such as the number of hidden units, are selected empirically to work well for this problem. Conceptually, the SimpleRNN layer processes inputs sequentially, passing a hidden state from one time step to the next. At each step, the current input and the previous hidden state are combined to produce a new hidden representation and an output.

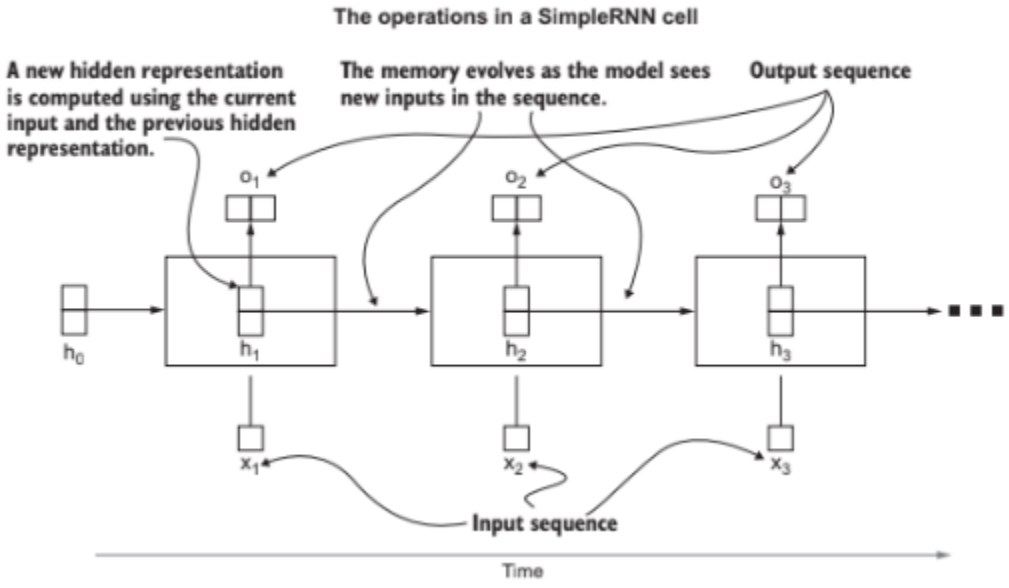



*Figure 4.20 The functionality of a SimpleRNN cell*

This figure illustrates how information flows through time in a SimpleRNN. The hidden state acts as memory, allowing the network to retain information about earlier inputs in the sequence. Because the hidden state depends on its previous value, these models are referred to as recurrent neural networks.

By default, Keras manages the propagation of hidden states automatically. Since this task only requires the final output of the sequence, the SimpleRNN layer can be directly connected to a Dense layer without exposing or manually handling hidden states.

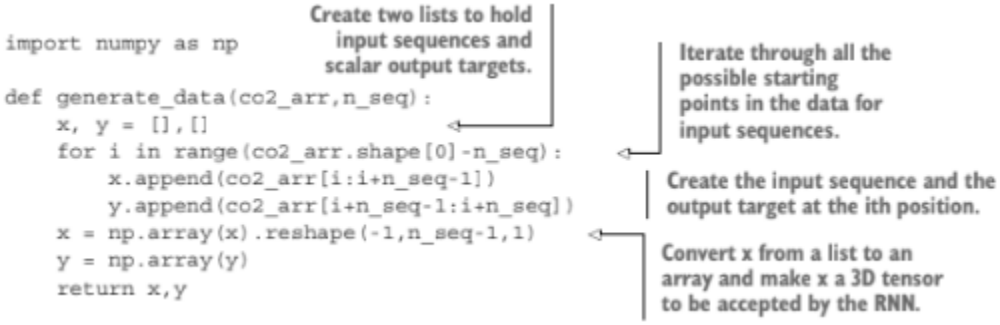
Although Keras allows layers to be built lazily without specifying input_shape, this can lead to errors. Explicitly defining the input shape in the first layer is generally safer, especially when working with sequential data.

Unlike earlier classification tasks, this problem is a **regression task**, as the model predicts continuous $CO_2$ concentration values rather than discrete classes. Consequently, **mean squared error (MSE)** is selected as the loss function, and the model is compiled using the Adam optimizer.

rnn.compile(loss='mse', optimizer='adam')

When training begins, an error occurs indicating that the SimpleRNN layer expects a **three-dimensional input**, but a two-dimensional tensor was provided. This highlights a key requirement of recurrent layers: input data must be structured as **(batch size, time steps, features)**. Even if only one feature is present, that dimension must still exist.

After identifying the issue, the input data is reshaped so that each sequence has an explicit feature dimension. The updated data generation function ensures that the input tensor conforms to the required three-dimensional format.




*Listing 4.5 The corrected generate_data() function with proper input shape*

Once corrected, the model trains successfully. The loss decreases steadily from an initial value of approximately 0.5 to around 0.015 by the final epoch, indicating that the network has learned meaningful temporal patterns from the data.

---

### 4.3.3 Predicting Future $CO_2$ Values with the Trained Model

Evaluating regression models is inherently more challenging than evaluating classification models, as there is no straightforward accuracy metric for continuous outputs. Instead of relying solely on numerical loss values, the trained model is used to **predict future $CO_2$ concentrations**, and the results are assessed visually.

The prediction process begins by extracting the last 12 values from the **Average Diff** column and reshaping them to match the expected input format. Since the model predicts relative changes rather than absolute values, the last known true $CO_2$ concentration is stored separately to reconstruct absolute predictions.

```
history = data["Average Diff"].values[-12:].reshape(1,-1,1)
true_vals = []
prev_true = data["Average"].values[-1]
for i in range(60):
    p_diff = rnn.predict(history).reshape(1,-1,1)
    history = np.concatenate((history[:,1:,:],p_diff),axis=1)
    true_vals.append(prev_true+p_diff[0,0,0])
    prev_true = true_vals[-1]
```

Save the very last absolute CO2 concentration value to compute the actual values from the relative predictions.

The first data sequence to start predictions from, which is reshaped to the correct shape the SimpleRNN accepts

Predict for the next 60 months.

Make a prediction using the data sequence.

Modify the history so that the latest prediction is included.

Compute the absolute CO2 concentration.

Update prev_true so that the absolute CO2 concentration can be computed in the next time step.

*Listing 4.6 Future $CO_2$ prediction logic using the trained model*

The model then recursively predicts the next 60 months (five years) of $CO_2$ concentration. After each prediction, the newly predicted value is appended to the input sequence, and the oldest value is removed. This rolling window approach allows the model to generate long-term forecasts by continuously feeding its own predictions back into the network.

Each predicted relative change is converted into an absolute $CO_2$ concentration by adding it to the previously known true value. This process is repeated iteratively, producing a sequence of future predictions.

When visualized, the predicted trend closely follows the historical upward trajectory of $CO_2$ levels while preserving seasonal fluctuations.

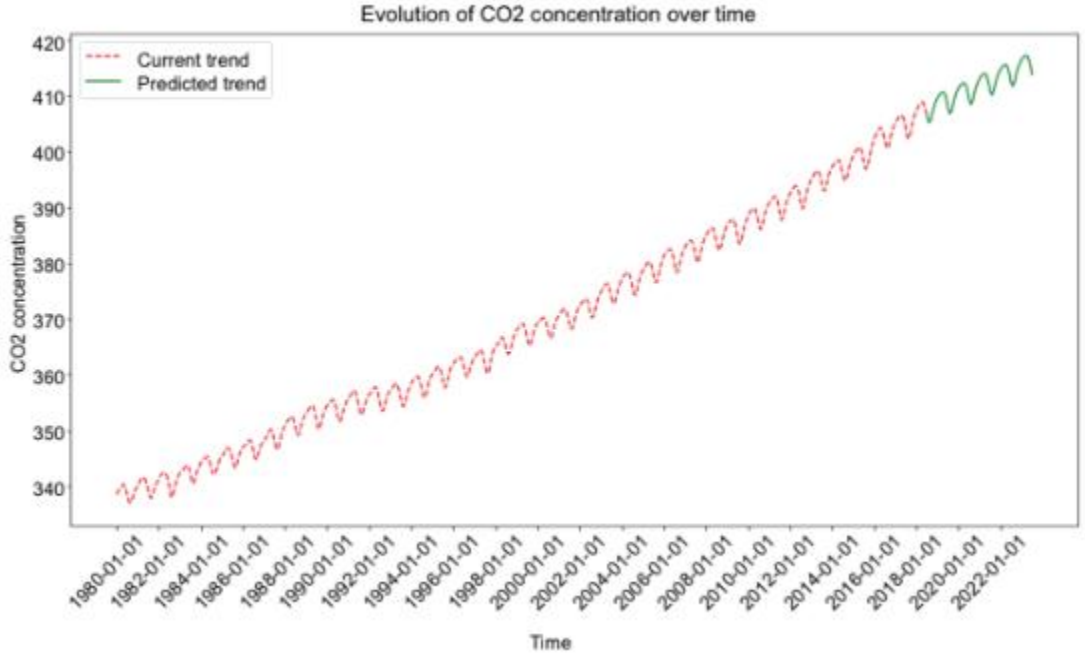


*Figure 4.21 $CO_2$ concentration predicted over the next five years*

Despite the simplicity of the model, the results are highly encouraging. The network successfully captures both the long-term increase and the cyclical behavior of atmospheric $CO_2$, demonstrating the effectiveness of RNNs for time-series forecasting.

**Summary**

This section demonstrates how recurrent neural networks can be implemented and trained for real-world time-series prediction tasks. By carefully preparing sequential data, selecting an appropriate model architecture, and respecting the strict input requirements of recurrent layers, a SimpleRNN is able to learn meaningful temporal patterns. The trained model produces realistic future $CO_2$ predictions, highlighting the practical value of RNNs for forecasting continuous environmental data and reinforcing their importance in deep learning applications involving sequences.