Lecture Book
B.Sc Image Processing

D. T. McGuiness, PhD

Version: 2024.0.0.2

Topics on Robotics & Vision

Contents

Part I.

AI in Image Processing

Chapter 1

Introduction to Artificial Neural Networks

Table of Contents

1.1 Introduction

It is quite apparent that life imitates life and engineers are inspired by nature. It seems only logical, then, to look at the brain's architecture for inspiration on how to build an intelligent machine. This is

Figure 1.1.: Nature always is a great source of inspiration for good design. For example, the beak of a bird is aerodynamically efficient and was used in designing the Bullet train [4].The field is of emulating models, systems, and elements of nature for the purpose of solving complex human problems is called biomimetics [40].

the logic that sparked Artificial Neural Networks (ANN)s, Machine Learning (ML) models inspired by the networks of biological neurons found in our brains. However, although planes were inspired by birds, they don't have to flap their wings to fly. Similarly, ANNs have gradually become quite different from their biological cousins. Some researchers even argue that we should drop the biological analogy altogether such as calling them units rather than neurons [1], as some consider this naming to decrease the amount of creativity we can give to the topic .

ANNs are at the very core of **deep learning**. They are versatile, powerful, and scalable, making them ideal to tackle large and highly complex ML tasks such as classifying billions of images (e.g., Google Images), powering speech recognition services (e.g., Apple's Siri), recommending the best videos to watch to hundreds of millions of users every day (e.g., YouTube), or learning to beat the world champion at the game of Go (DeepMind's AlphaGo [18]).

The will treat this chapter as a formal introduction to ANN, starting with a tour of the very first ANN architectures and leading up to multilayer perceptrons, which are heavily used today.

In the second part, we will look at how to implement neural networks using TensorFlow's Keras API. This is a beautifully designed and simple high-level API for building, training, evaluating, and running neural networks. While it may look simple at first glance, it is expressive and flexible enough to let you build a wide variety of neural network architectures.

> For most of your use cases, using `keras` will be enough.

Figure 1.2.: The prolific advancements of computers and neural networks have allowed us to tackle problems once deemed impossible. A game of GO requires uncountable amount of moves, yet using ML it was possible to create a software capable of beating the world champion.

1.2 From Biology to Silicon: Artificial Neurons

While it may seem they are the cutting edge in ML, ANNs have been around for quite a while: they were first introduced back in 1943 by the neurophysiologist Warren McCulloch and the mathematician Walter Pitts. In their landmark paper *A Logical Calculus of Ideas Immanent in Nervous Activity*, They presented a simplified computational model of how biological neurons might work together in animal brains to perform complex computations using propositional logic. This was the first artificial neural network architecture [29].

Since then many other architectures have been invented. The early successes of ANNs led to the widespread belief that we would soon be conversing with truly intelligent machines. When it became clear in the 1960s that this promise would go unfulfilled (at least for quite a while), funding flew elsewhere, and ANNs entered a long winter. This is also known as the 1st AI winter [19]. In the early 1980s, new architectures were invented and better training techniques were developed, sparking a revival of interest in connectionism, the study of neural networks. But progress was slow, and by the 1990s other powerful ML techniques had been invented, such as Support Vector Machines (SVM). These techniques seemed to offer better results and stronger theoretical foundations than ANNs, so once again the study of neural networks was put on hold and this is known as the 2nd AI winter.

We are now witnessing yet another wave of interest in ANNs. Will this wave die out like the previous ones did? Well, here are a few good reasons to believe that this time is different and that the renewed interest in ANNs will have a much more profound impact on our lives:

There is now a **huge quantity of data available** to train neural networks, and ANNs frequently outperform other ML techniques on very large and complex problems. One of the major turning points of ANN was the fundamental question of:

Is our understanding of the model at fault or is it merely the lack of data to train?

The tremendous increase in computing power since the 1990s now makes it possible to train large neural networks in a reasonable amount of time. This is in part due to **Moore's law**, but also thanks to the gaming industry, which has stimulated the production of powerful GPU cards by the millions which have become the norm to train ML instead of CPUs.

Moore's Law

the number of components in integrated circuits has doubled about every 2 years over the last 50 years.

In addition to previous additions, cloud platforms have made this power accessible to everyone. The training algorithms have been improved. To be fair they are only slightly different from the ones used in the 1990s, but these relatively small tweaks have had a huge positive impact.

Some theoretical limitations of ANNs have turned out to be benign in practice. For example, many people thought that ANN training algorithms were doomed because they were likely to get stuck in local optima [11], but it turns out that this is not a big problem in practice, especially for larger neural networks: the local optima often perform almost as well as the global optimum.

ANNs seem to have entered a virtuous circle of funding and progress. Amazing products based on ANNs regularly make the headline news, which pulls more and more attention and funding toward them, resulting in more and more progress and even more amazing products.

1.2.1 Biological Neurons

Before we discuss artificial neurons, let's take a quick look at a biological neuron. It is an unusual-looking cell mostly found in animal brains.

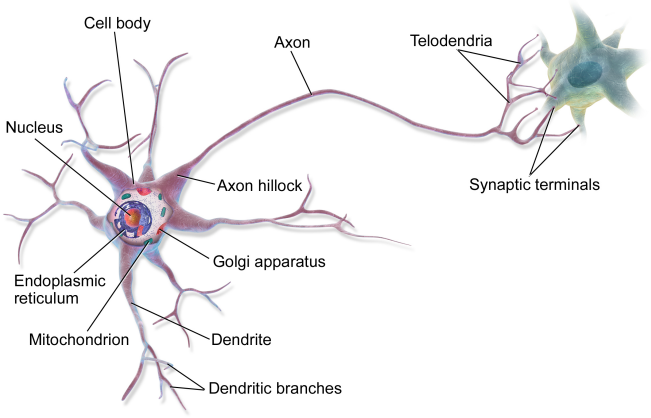

Figure 1.3.: A neuron or nerve cell is an excitable cell that fires electric signals called action potentials across a neural network in the nervous system. Neurons communicate with other cells via synapses, which are specialized connections that commonly use minute amounts of chemical neurotransmitters to pass the electric signal from the presynaptic neuron to the target cell through the synaptic gap [35].

It's composed of a cell body containing the nucleus and most of the cell's complex components, many branching extensions called dendrites, plus one very long extension called the axon. The axon's length may be just a few times longer than the cell body, or up to tens of thousands of times longer.
Near its extremity the axon splits off into many branches called *telodendria*, and at the tip of these branches are minuscule structures called synaptic terminals (or simply synapses), which are connected to the dendrites or cell bodies of other neurons. Biological neurons produce short electrical impulses called action potentials (APs, or just signals), which travel along the axons and make the synapses release chemical signals called neurotransmitters. When a neuron receives a sufficient amount of these neurotransmitters within a few milliseconds, it fires its own electrical impulses (actually, it depends on the neurotransmitters, as some of them inhibit the neuron from firing).

Therefore, individual biological neurons seem to behave in a simple way, but they're organized in a vast network of billions, with each neuron typically connected to thousands of other neurons. Highly complex computations can be performed by a network of fairly simple neurons, much like a complex anthill can emerge from the combined efforts of simple ants. The architecture of biological neural networks (BNNs) is the subject of active research, but some parts of the brain have been mapped [3]. These efforts show that neurons are often organized in consecutive layers, especially in the cerebral cortex (the outer layer of the brain).

1.2.2 Logical Computations with Neurons

McCulloch and Pitts proposed a very simple model of the biological neuron, which later became known as an **artificial neuron**: it has one or more binary (on/off) inputs and one binary output. The artificial neuron activates its output when more than a certain number of its inputs are active. In their paper, McCulloch and Pitts showed that even with such a simplified model it is possible to build a network of artificial neurons that can compute any logical proposition you want. To see how such a network works, let's build a few ANNs that perform various logical computationsm, assuming that a neuron is activated when at least two of its input connections are active. Let's see what these networks do:

- The first network on the left is the identity function: if neuron **A** is activated, then neuron **C** gets

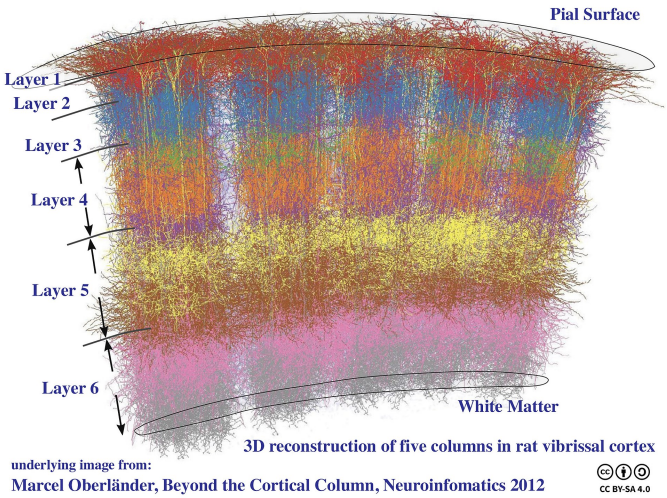

Figure 1.4.: A cortical column is a group of neurons forming a cylindrical structure through the cerebral cortex of the brain perpendicular to the cortical surface. The structure was first identified by Vernon Benjamin Mountcastle in 1957. He later identified minicolumns as the basic units of the neocortex which were arranged into columns. Each contains the same types of neurons, connectivity, and firing properties. Columns are also called hypercolumn, macrocolumn, functional column or sometimes cortical module

activated as well (since it receives two input signals from neuron **A**); but if neuron **A** is off, then neuron **C** is off as well.

- The second network performs a logical **AND**: neuron **C** is activated only when both neurons **A** and **B** are activated (a single input signal is not enough to activate neuron **C**).
- The third network performs a logical **OR**: neuron **C** gets activated if either neuron **A** or neuron **B** is activated (or both).
- Finally, if we suppose that an input connection can inhibit the neuron's activity (which is the case with biological neurons), then the fourth network computes a slightly more complex logical proposition: neuron C is activated only if neuron A is active and neuron B is off. If neuron A is active all the time, then you get a logical NOT: neuron C is active when neuron B is off, and vice versa.

You can imagine how these networks can be combined to compute complex logical expressions.

1.2.3 The Perceptron

The perceptron is one of the simplest ANN architectures, invented in 1957 by Frank Rosenblatt [6]. It is based on a slightly different artificial neuron called a Threshold Logic Unit (TLU), or sometimes a Linear Threshold Unit (LTU) which can be seen in Fig. 1.5. The inputs and output are numbers (this is instead of binary on/off values), and each input connection is associated with a weight. The TLU first computes a linear function of its inputs:

$$z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b = \mathbf{x}^\mathsf{T}\mathbf{w} + b$$

Then it applies a **step function** to the result:

$$h(x) = \text{step}(z) \qquad \text{where} \qquad z = \mathbf{x}^\mathsf{T}\mathbf{w}.$$

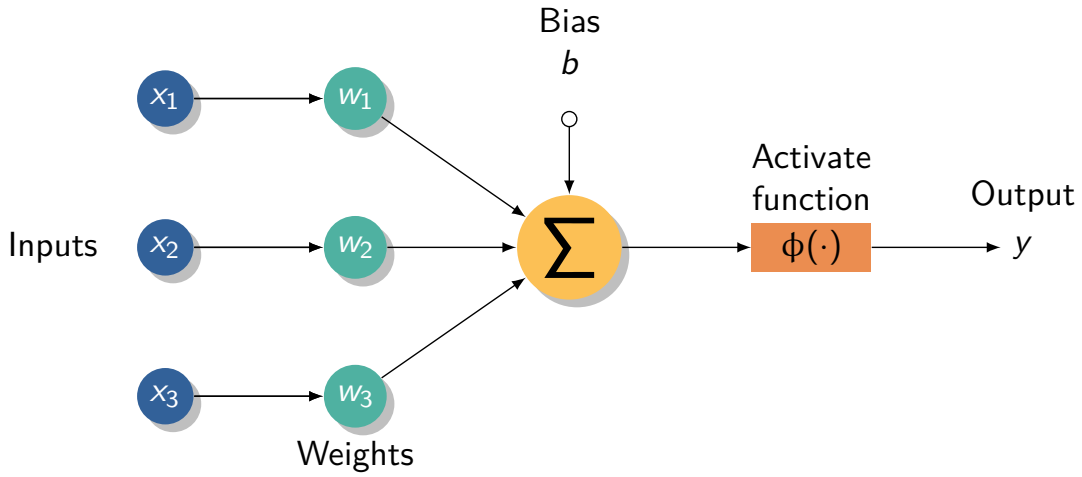

Figure 1.5.: Threshold logic unit: an artificial neuron which computes a weighted sum of its inputs then applies a certain activation function.

It is similar to logistic regression, except it uses a step function instead of the logistic function. Just like in logistic regression, the model parameters are the input weights w and the bias term b.
The most common step function used in perceptrons is the **Heaviside step** and sometimes the **sign function** is used instead [36].

$$\text{heaviside}(z) = \begin{cases} 0 & \text{if} \quad z < 0, \\ 1 & \text{if} \quad z \geq 0. \end{cases} \qquad \text{sgn}(z) = \begin{cases} -1 & \text{if} \quad z < 0, \\ 0 & \text{if} \quad z = 0, \\ +1 & \text{if} \quad z > 0, \end{cases}$$

A single TLU can be used for simple **linear binary classification**:

> *It computes a linear function of its inputs, and if the result exceeds a threshold, it outputs the positive class. Otherwise, it outputs the negative class. They exhibit a similar behaviour to logistic regression or linear SVM classification.*

It is possible, for example, use a single TLU to classify iris flowers [8] (a famous dataset used by statisticians and ML researchers) based on petal length and width. Training such a TLU would require finding the right values for w_1, w_2, b.

A perceptron is composed of one or more TLUs organized in a single layer, where every TLU is connected to every input. Such a layer is called a **fully connected layer**, or a dense layer. The inputs constitute the input layer and since the layer of TLUs produces the final outputs, it is called the output layer.

This perceptron can classify instances simultaneously into three (3) different binary classes, which makes it a **multilabel classifier**. It may also be used for multiclass classification.

Using linear algebra, the following equation can be used to efficiently compute the outputs of a layer of artificial neurons for several instances at once.

$$h_{\mathbf{W},\mathbf{b}}(\mathbf{X}) = \phi(\mathbf{XW} + \mathbf{b})$$

In this equation:

- **X** represents the matrix of input features. It has one row per instance and one column per feature.
- **W** is the weight matrix containing all the connection weights. It has one row per input and one column per neuron.
- **b** is the bias term containing all the bias terms: one per neuron.
- ϕ is the activation function is called the activation function: when the artificial neurons are TLUs, it is a step function.

Now the question is:

How is this perceptron train?

The original perceptron training algorithm proposed by Rosenblatt was largely inspired by Hebb's rule [37]. In his 1949 book *The Organization of Behaviour*, Donald Hebb suggested that when a biological neuron triggers another neuron often, the connection between these two neurons grows stronger [14].

Siegrid Löwel later summarized Hebb's idea in the catchy phrase,

Cells that fire together, wire together

This means the connection weight between two neurons tends to increase when they fire simultaneously.

> This rule later became known as Hebb's rule (or Hebbian learning [10])

Perceptrons are trained using a variant of this rule that takes into account the error made by the network when it makes a prediction. The perceptron learning rule **reinforces connections that help reduce the error**.

More specifically, the perceptron is fed one training instance at a time, and for each instance it makes its predictions. For every output neuron that produced a wrong prediction, it reinforces the connection weights from the inputs that would have contributed to the correct prediction.

$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta \left(y_j - \hat{y}_j \right) x_i x$$

where:

- $w_{i,j}$ is the connection weight between the i^{th} input and the j^{th} neuron.
- x_i is the i^{th} input value of the current training instance.
- $\hat{y}^j$ is the output of the j^{th} output neuron for the current training instance.
- y^j is the target output of the j^{th} output neuron for the current training instance.
- η is the learning rate.

The decision boundary of each output neuron is **linear**, therefore perceptrons are incapable of learning complex patterns. However, if the training instances are linearly separable, Rosenblatt demonstrated that this algorithm would converge to a solution.

This is called the perceptron convergence theorem.

C.R.1 python
```python
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron
iris = load_iris(as_frame=True)
X = iris.data[["petal length (cm)", "petal width (cm)"]].values y = (iris.target == 0) # Iris setosa
per_clf = Perceptron(random_state=42) per_clf.fit(X, y)
X_new = [[2, 0.5], [3, 1]]
y_pred = per_clf.predict(X_new) # predicts True and False for these 2 flowers
```

For those of you who have taken a **Data Science II** course, you may have noticed that the perceptron learning algorithm strongly resembles *stochastic gradient descent*. In fact, `sklearn`'s `Perceptron` class is equivalent to using an `SGDClassifier` with the following hyperparameters:

- `loss="perceptron"`,
- `learning_rate="constant"`,
- `eta0=1` (the learning rate),
- `penalty=None` (no regularization).

In their 1969 monograph Perceptrons, Marvin Minsky and Seymour Papert highlighted a number of serious weaknesses of perceptrons: in particular, they are incapable of solving some trivial problems (e.g., the exclusive OR (XOR) classification problem).

XOR Problem

A simple logic gate problem which is proven to be unsolvable using a single-layer perceptron.

This is true of any other linear classification model, but researchers had expected much more from perceptrons, and some were so disappointed, they dropped neural networks altogether in favour of higher-level problems such as logic, problem solving, and search. The lack of practical applications also didn't help.

It turns out that some of the limitations of perceptrons can be eliminated by stacking multiple perceptrons. The resulting ANN is called a MLP and a MLP can solve the XOR problem [32].

Perceptrons **DO NOT** output a class probability. This is one reason to prefer logistic regression over perceptrons. Moreover, perceptrons do not use any regularization by default, and training stops as soon as there are no more prediction errors on the training set, so the model typically does not generalize as well as logistic regression or a linear SVM classifier. However, perceptrons may train a bit faster.

1.2.4 Multilayer Perceptron and Backpropagation

An MLP is composed of one input layer, one or more layers of TLUs called **hidden layers**, and one final layer of TLUs called the output layer. The layers close to the input layer are usually called the

lower layers, and the ones close to the outputs are usually called the upper layers.

> The signal flows only in one direction (inputs to outputs), so this architecture is an example of a Feedforward Neural Networks (FNN) [5].

When an ANN contains a deep stack of hidden layers, it is called a Deep Neural Networks (DNN). The field of deep learning studies DNNs, and more generally it is interested in models containing deep stacks of computations [34].

For many years researchers struggled to find a way to train MLPs, without success. In the early 1960s

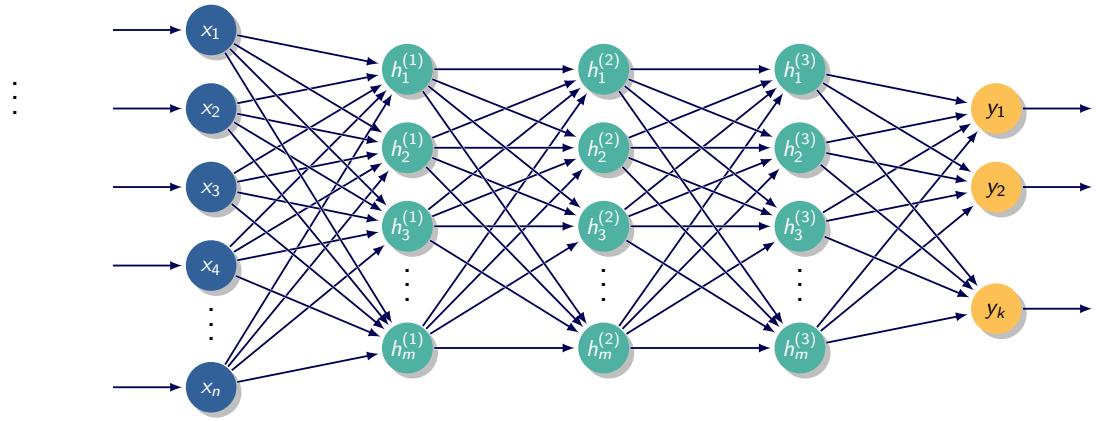

Figure 1.6.: Architecture of a Multilayer Perceptron with five inputs, three hidden layer of four neurons, and three output neurons.

several researchers discussed of using gradient descent to train neural networks. This requires computing the gradients of the model's error with regard to the model parameters and at that time, it wasn't clear at the time how to do this efficiently with such a complex model containing so many parameters.

Then, in 1970, a researcher named *Seppo Linnainmaa* introduced in his master's thesis a technique to compute all the gradients automatically and efficiently. This algorithm is now called **reverse-mode automatic differentiation** [27]. In just two passes through the network (one forward, one backward), it is able to compute the gradients of the neural network's error with regard to every single model parameter.

In other words, it can find out how each connection weight and each bias should be tweaked in order to reduce the neural network's error. These gradients can then be used to perform a gradient descent step. Repeating the process of computing the gradients automatically and taking a gradient descent step, the neural network's error will gradually drop until it eventually reaches a minimum.

This combination of reverse-mode automatic differentiation and gradient descent is now called **backpropagation** [15].

> There are various automatic differentiation techniques (i.e., forward and reverse), with each having its own advantages and disadvantages. Reverse-mode autodiff is well suited when the function to differentiate has many variables (e.g., connection weights and biases) and few

outputs (e.g., one loss).

Backpropagation can actually be applied to all sorts of computational graphs, not just neural networks: Linnainmaa's M.Sc thesis was not about neural nets, it was more general. It was several more years before backprop started to be used to train neural networks, but it still wasn't mainstream.

Then, in 1986, David Rumelhart, Geoffrey Hinton, and Ronald Williams published a groundbreaking paper analyzing how backpropagation allowed neural networks to learn useful internal representations [33]. Their results were so impressive that backpropagation was quickly popularized in the field. Today, it is by far the most popular training technique for neural networks.

Let's run through how backpropagation works again in a bit more detail:

1. It handles one mini-batch at a time, and goes through the full training set multiple times. Each pass is called an **epoch**.
2. Each mini-batch enters the network through the input layer. The algorithm then computes the output of all the neurons in the first hidden layer, for every instance in the mini-batch. The result is passed on to the next layer, its output is computed and passed to the next layer, and so on until we get the output of the last layer, the output layer.
3. Next, the algorithm measures the network's output error. This means, it uses a loss function that compares the desired output and the actual output of the network, and returns some measure of the error.
4. It then computes how much each output bias and each connection to the output layer contributed to the error. This is done analytically by applying the chain rule, which makes this step fast and precise.
5. The algorithm then measures how much of these error contributions came from each connection in the layer below, again using the chain rule, working backward until it reaches the input layer. As explained earlier, this reverse pass efficiently measures the error gradient across all the connection weights and biases in the network by propagating the error gradient backward through the network.
6. Finally, the algorithm performs a gradient descent step to tweak all the connection weights in the network, using the error gradients it just computed.

Initialize all the hidden layers' connection weights randomly, or training will fail.

For example, if you initialize all weights and biases to zero, then all neurons in a given layer will be perfectly identical, and therefore backpropagation will affect them in exactly the same way, so they will remain identical.

In other words, despite having hundreds of neurons per layer, your model will act as if it had only one neuron per layer: it won't be too smart. If instead you randomly initialize the weights, you **break the symmetry** and allow back-propagation to train a diverse team of neurons.

In short, backpropagation makes predictions for a mini-batch (forward pass), measures the error, then goes through each layer in reverse to measure the error contribution from each parameter (reverse pass), and finally tweaks the connection weights and biases to reduce the error, which is the gradient

descent step.

For back-propagation to work properly, Rumelhart and his colleagues made a key change to the MLP's architecture by replacing the step function with the logistic function:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Which is also called the **sigmoid function**. This was an important improvement as step function contains only flat segments, so there is no gradient to work with, while the sigmoid function has a well-defined nonzero derivative everywhere. In fact, the backpropagation algorithm works well with many other activation functions, not just the sigmoid function.

Here are two (2) other popular choices:

The hyperbolic tangent function Model

$$\tanh(z) = 2\sigma(2z) - 1$$

Similar sigmoid function, this activation function is also S-shaped, continuous, and differentiable, but its output value ranges from –1 to 1, instead of 0 to 1 like the sigmoid function.

This bigger range tends to make each layer's output more or less centered around 0 at the beginning of training, which often helps speed up convergence.

The rectified linear unit function Model

$$ReLU(z) = max(0, z)$$

It is continuous but unfortunately not differentiable at $z = 0$ as the slope changes abruptly, which can make gradient descent bounce around, and its derivative is 0 for z < 0.

In practice, however, it works very well and has the advantage of being fast to compute, so it has become the default.

Importantly, the fact that it does not have a maximum output value helps reduce some issues during gradient descent.

You might wonder what is the point of an activation function, let alone whether it is linear or not? Chaining several linear transformations, gives you only linear transformation. For example:

$$f(x) = 2x + 3 \quad \text{and} \quad g(x) = 5x - 1 \quad \rightarrow \quad f(g(x)) = 2(5x - 1) + 3 = 10x + 1$$

You don't have some nonlinearity between layers, then even a deep stack of layers is equivalent to a single layer, and you can't solve very complex problems with that.

> A large enough DNN with nonlinear activations can theoretically approximate any continuous function.

1.2.5 Regression MLPs

First, MLPs can be used for regression tasks. If you want to predict a single value (e.g., the price of a house, given many of its features), you just need a single output neuron:

its output is the predicted value

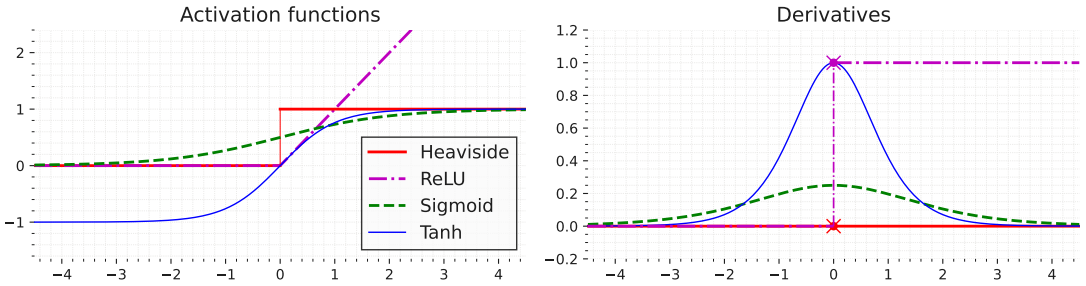

Figure 1.7.: The activation function of a node in an ANN is a function which calculates the output of the node based on its individual inputs and their weights. Nontrivial problems can be solved using only a few nodes if the activation function is nonlinear [23]. Modern activation functions include the smooth version of the ReLU, the GELU, which was used in the 2018 BERT model [16], the logistic (sigmoid) function used in the 2012 speech recognition model developed by Hinton et al [17], the ReLU used in the 2012 AlexNet computer vision model [24] and in the 2015 ResNet model.

For multivariate regression (i.e., to predict multiple values at once), you need one output neuron per output dimension. As an example, to locate the center of an object in an image, you need to predict 2D coordinates, so you need two (2) output neurons. If you also want to place a bounding box around the object, then you need two more numbers: the width and the height of the object.

So, in the end you end up with four (4) output neurons.

`sklearn` includes an `MLPRegressor` class, so let's use it to build an MLP with three hidden layers composed of 50 neurons each, and train it on the California housing dataset.

For simplicity, we will use `sklearn`'s `fetch_california_housing()` function to load the data instead of downloading from a sketchy website.

The following code starts by fetching and splitting the dataset, then it creates a pipeline to standardise the input features before sending them to the `MLPRegressor`. This is very important for neural networks as they are trained using gradient descent, and gradient descent does not converge very well when the features have very different scales.

Finally, the code trains the model and evaluates its validation error. The model uses the ReLU activation function in the hidden layers, and it uses a variant of gradient descent called Adam to minimize the mean squared error, with a little bit of ℓ_2 regularisation:

C.R. 2
python
```
from sklearn.datasets import fetch_california_housing
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPRegressor
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

housing = fetch_california_housing()
X_train_full, X_test, y_train_full, y_test = train_test_split(
    housing.data, housing.target, random_state=42)
X_train, X_valid, y_train, y_valid = train_test_split(
```

C.R. 3 python
```
    X_train_full, y_train_full, random_state=42)

mlp_reg = MLPRegressor(hidden_layer_sizes=[50, 50, 50], random_state=42)
pipeline = make_pipeline(StandardScaler(), mlp_reg)
pipeline.fit(X_train, y_train)
y_pred = pipeline.predict(X_valid)
rmse = mean_squared_error(y_valid, y_pred, squared=False)
```

We get a validation RMSE of about 0.505, which is comparable to what you would get with a random forest classifier.

> This MLP does not use any activation function for the output layer, so it's free to output any value it wants.

This is generally fine, but if you want to guarantee that the output will always be positive, then you should use the ReLU activation function in the output layer, or the softplus activation function, which is a smooth variant of ReLU: softplus(z) = log(1 + exp(z)).

Softplus is close to 0 when z is negative, and close to z when z is positive. Finally, if you want to guarantee that the predictions will always fall within a given range of values, then you should use the sigmoid function or the hyperbolic tangent, and scale the targets to the appropriate range: 0 to 1 for sigmoid and –1 to 1 for tanh.

Sadly, the `MLPRegressor` class does not support activation functions in the output layer.

> Building and training a standard MLP with `sklearn` is very convenient, but features are limited. This is why we will switch to Keras in the second part of this chapter.

The `MLPRegressor` class uses the mean squared error, which is usually what you want for regression, but if you have a lot of outliers in the training set, you may prefer to use the mean absolute error instead. Alternatively, you may want to use the Huber loss, which is a combination of both. It is quadratic when the error is smaller than a threshold □ (typically 1) but linear when the error is larger than □. The linear part makes it less sensitive to outliers than the mean squared error, and the quadratic part allows it to converge faster and be more precise than the mean absolute error. However, `MLPRegressor` only supports the MSE.

1.2.6 Classification MLPs

MLPs can also be used for **classification** tasks. For a binary classification problem, you just need a single output neuron using the sigmoid activation function: the output will be a number between 0 and 1, which you can interpret as the estimated probability of the positive class.

> The estimated probability of the negative class is equal to one minus that number.

MLPs can also easily handle multilabel binary classification tasks. For example, you could have an email classification system that predicts whether each incoming email is ham or spam, and simultaneously predicts whether it is an urgent or nonurgent email.

In this case, you would need two output neurons, both using the sigmoid activation function: the first would output the probability that the email is spam, and the second would output the probability that it is urgent. More generally, you would dedicate one output neuron for each positive class. Note that the output probabilities do not necessarily add up to 1. This lets the model output any combination of labels: you can have nonurgent ham, urgent ham, nonurgent spam, and perhaps even urgent spam (although that would probably be an error).

If each instance can belong only to a single class, out of three or more possible classes (e.g., classes 0 through 9 for digit image classification), then you need to have one output neuron per class, and you should use the softmax activation function for the whole output layer (see Figure 10-9). The softmax function (introduced in Chapter 4) will ensure that all the estimated probabilities are between 0 and 1 and that they add up to 1, since the classes are exclusive. As you saw in Chapter 3, this is called multiclass classification.

Regarding the loss function, since we are predicting probability distributions, the cross-entropy loss (or x-entropy or log loss for short, see Chapter 4) is generally a good choice.

`sklearn` has an `MLPClassifier` class in the `sklearn.neural_network` package. It is almost identical to the `MLPRegressor` class, except that it minimizes the cross entropy rather than the MSE. Give it a try now, for example on the iris dataset. It's almost a linear task, so a single layer with 5 to 10 neurons should suffice (make sure to scale the features).

1.3 Implementing MLPs with Keras

Keras is TensorFlow's high-level deep learning API: it allows you to build, train, evaluate, and execute all sorts of neural networks. The original Keras 12 library was developed by François Chollet as part of a research project and was released as a standalone open source project in March 2015. It quickly gained popularity, owing to its ease of use, flexibility, and beautiful design.

Application

Keras used to support multiple backends, including TensorFlow, PlaidML, Theano, and Microsoft Cognitive Toolkit (CNTK) (the last two are sadly deprecated), but since version 2.4, Keras is TensorFlow-only. Similarly, TensorFlow used to include multiple high-level APIs, but Keras was officially chosen as its preferred high-level API when TensorFlow 2 came out. Installing TensorFlow will automatically install Keras as well, and Keras will not work without TensorFlow installed. In short, Keras and TensorFlow fell in love and got married. Other popular deep learning libraries include PyTorch by Facebook and JAX by Google.13

1.3.1 Building an Image Classifier Using Sequential API

Before we do anything, we need to load a dataset. We will use Fashion MNIST. There are 70,000 grayscale images of 28 × 28 pixels each, with 10 classes where images represent fashion items rather than handwritten digits, so each class is more diverse, and the problem turns out to be significantly challenging.

Using Keras to load the dataset

`keras` provides utility functions to fetch and load common datasets, including MNIST, Fashion MNIST, and a few more.

Let's load Fashion MNIST. It's already shuffled and split into a training set (60,000 images) and a test set (10,000 images), but we'll hold out the last 5,000 images from the training set for validation:

C.R. 4 python
```python
import tensorflow as tf

fashion_mnist = tf.keras.datasets.fashion_mnist.load_data()
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist
X_train, y_train = X_train_full[:-5000], y_train_full[:-5000]
X_valid, y_valid = X_train_full[-5000:], y_train_full[-5000:]
```

> TensorFlow is usually imported as `tf`, and the Keras API is available via `tf.keras`.

> When loading MNIST or Fashion MNIST using `tf.keras` rather than `sklearn`, an important difference is that every image is represented as a 28-by-28 array rather than a 1D array of size 784 with intensities are represented as integers (from 0 to 255) rather than floats (from 0.0 to 255.0).

Let's take a look at the shape and data type of the training set:

C.R. 5 python
```python
print("The size of the training dataset: ", X_train.shape)
print("The type of the training dataset: ", X_train.dtype)
```

text
```
The size of the training dataset:  (55000, 28, 28)
The type of the training dataset:  uint8
```

To make it simple, we'll scale the pixel intensities down to the 0–1 range by dividing them by 255.0

> This operation also converts the integer values to floats.

C.R. 6 python
```python
X_train, X_valid, X_test = X_train / 255., X_valid / 255., X_test / 255.
```

Using Fashion MNIST, we need the list of class names to know what we are dealing with:

C.R. 7 python
```python
class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",
               "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]
```

For example, the first image in the training set represents an ankle boot: and below we can see some examples of the Fashion MNIST dataset.

1.3.2 Creating the model using the sequential API

It is time to build the neural network. Here is a classification MLP with two (2) hidden layers:

Figure 1.8.: An example of a data within the Fashion MNIST.

Figure 1.9.: A random collection of dataset, making the Fashion MNIST.

C.R. 8
python
```
tf.random.set_seed(42)
model = tf.keras.Sequential()
model.add(tf.keras.layers.InputLayer(shape=[28, 28]))
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(300, activation="relu"))
model.add(tf.keras.layers.Dense(100, activation="relu"))
model.add(tf.keras.layers.Dense(10, activation="softmax"))
```

Let's try to understand the code:

1. Set `tf` random seed to make the results reproducible: the random weights of the hidden layers and the output layer will be the same every time you run your code. You could also choose to use the `tf.keras.utils.set_random_seed()` function, which conveniently sets the random seeds for TensorFlow, Python (`random.seed()`), and NumPy (`np.random.seed()`).

2. Next line creates a *Sequential model*. This is the simplest kind of Keras model for neural networks, composed of a single stack of layers connected sequentially. This is called the sequential API.

3. We build the first layer (an Input layer) and add it to the model. We specify the input shape, which doesn't include the batch size, only the shape of the instances. Keras needs to know the shape of the inputs so it can determine the shape of the connection weight matrix of the first hidden layer.

4. We add a Flatten layer. Its role is to convert each input image into a 1D array: for example, if it receives

a batch of shape [32, 28, 28], it will reshape it to [32, 784]. In other words, if it receives input data X, it computes `X.reshape(-1, 784)`. This layer doesn't have any parameters; it's just there to do some simple pre-processing.

5. We add a Dense hidden layer with 300 neurons. It will use the ReLU activation function. Each Dense layer manages its own weight matrix, containing all the connection weights between the neurons and their inputs. It also manages a vector of bias terms (one per neuron).
6. We add a second Dense hidden layer with 100 neurons, also using the ReLU activation function.
7. We add a Dense output layer with 10 neurons (one per class), using the softmax activation function because the classes are exclusive.

> Writing the argument `activation="relu"` is equivalent to specifying `activation=tf.keras.activations.relu`. Other activation functions are available in the `tf.keras.activations` package.

Instead of adding the layers one by one as we just did, it's often more convenient to pass a list of layers when creating the Sequential model. You can also drop the Input layer and instead specify the `input_shape` in the first layer:

C.R. 9 python
```
tf.keras.backend.clear_session()
tf.random.set_seed(42)

model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    tf.keras.layers.Dense(300, activation="relu"),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(10, activation="softmax")
])
```

The model's `summary()` method displays all the model's layers, including each layer's name, which is automatically generated, its output shape, and its number of parameters.

The summary ends with the total number of parameters, including trainable and non-trainable parameters. Here we only have trainable parameters:

C.R. 10 python
```
tf.keras.utils.plot_model(model, imagePath+"mnist-_model.pdf", show_shapes=True)
```

Dense layers often have a lot of parameters. For example, the first hidden layer has 784-by-300 connection weights, with 300 bias terms, which adds up to 235,500 parameters.

This gives the model quite a lot of flexibility to fit the training data, but it also means that the model runs the risk of **over-fitting**, especially when you do not have a lot of training data.

Each layer in a model must have a unique name (e.g., `dense_2`). You can set the layer names explicitly using the constructor's name argument, but generally it's simpler to let Keras name the layers automatically, as we just did. Keras takes the layer's class name and converts it to snake case (i.e., a layer from the `MyCoolLayer` class is named `my_cool_layer` by default). Keras also ensures that the name

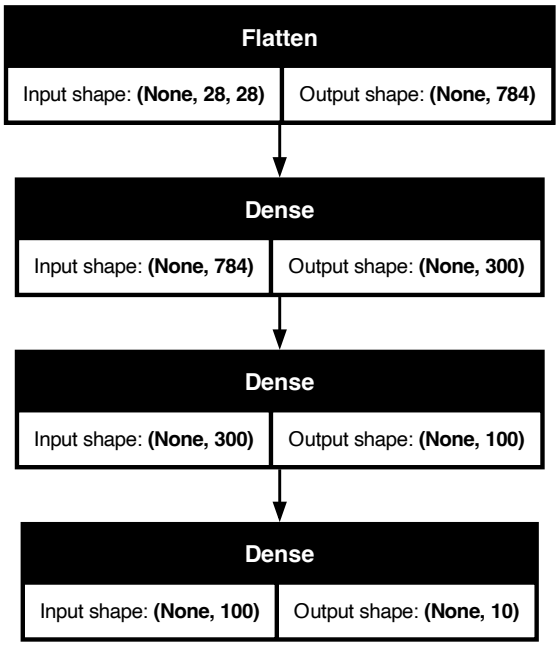

Figure 1.10.: The plot of the neural network, showcasing its layers.

is **globally unique**, even across models, by appending an index if needed, as in `dense_2`.

This naming scheme makes it possible to merge models easily without getting name conflicts.

> All global state managed by Keras is stored in a Keras session, which you can clear using `tf.keras.backend.clear_session()`.

You can easily get a model's list of layers using the layers attribute, or use the `get_layer()` method to access a layer by name:

C.R. 11
```python
print(model.layers)
```

C.R. 12
```text
[<Flatten name=flatten, built=True>,
<Dense name=dense, built=True>,
<Dense name=dense_1, built=True>,
<Dense name=dense_2, built=True>]
```

All the parameters of a layer can be accessed using its `get_weights()` and `set_weights()` methods.

For a Dense layer, this includes both the connection weights and the bias terms:

C.R. 13 python

```python
hidden1 = model.layers[1]
weights, biases = hidden1.get_weights()
print(weights)
```

C.R. 14 text

```text
[[-0.05415904  0.00010975 -0.00299759 ...  0.05136904  0.0740822
   0.06472497]
 [ 0.05510217 -0.01353022 -0.00363479 ...  0.07100512 -0.04926914
  -0.02905609]
 [-0.07024231  0.02524897 -0.04784295 ... -0.0521326   0.05084455
  -0.06636713]
 ...
 [ 0.0067075  -0.00256791 -0.064556   ...  0.05266081  0.03520959
  -0.02309504]
 [ 0.05826265 -0.0361187  -0.04228947 ...  0.05612285 -0.03179397
   0.06843598]
 [ 0.06636336 -0.00123435 -0.00247347 ...  0.01809192  0.03434542
   0.00700523]]
```

Notice that the Dense layer initialized the connection weights randomly.

> This is needed to break symmetry.

The biases were initialized to zeros, which is fine.

C.R. 15 python

```python
print(biases)
```

C.R. 16 text

```text
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

If you want to use a different initialization method, you can set `kernel_initializer` or `bias_initializer` when creating the layer.

Weight Matrix Shape

The shape of the weight matrix depends on the number of inputs, which is why we specified the `input_shape` when creating the model. If you do not specify the input shape, it's OK: Keras will simply wait until it knows the input shape before it actually builds the model parameters. This will happen either when you feed it some data (e.g., during training), or when you call its build() method. Until the model parameters are built, you will not be able to do certain things,

such as display the model summary or save the model. So, if you know the input shape when creating the model, it is best to specify it.

Model Compiling

After a model is created, we need to call its `compile()` method to specify the loss function and the optimizer to use, or you can specify a list of extra metrics to compute during training and evaluation:

C.R. 17 python
```
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="sgd",
              metrics=["accuracy"])
```

Before continuing, we need to explain what is going on here.

We use the `sparse_categorical_crossentropy` loss because we have sparse labels (i.e., for each instance, there is just a target class index, from 0 to 9 in this case), and the classes are **exclusive**.

- If we had one target probability per class for each instance (such as one-hot vectors, e.g., [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.] to represent class 3), then we would need to use the `categorical_crossentropy` loss instead.
- If we were doing binary classification or multilabel binary classification, then we would use the sigmoid activation function in the output layer instead of the softmax activation function, and we would use the `binary_crossentropy` loss.

Regarding the optimizer, `sgd` means that we will train the model using stochastic gradient descent. Keras will perform the backpropagation algorithm described earlier (i.e., reverse-mode autodiff plus gradient descent).

Finally, as this is a classifier, it's useful to measure its accuracy during training and evaluation, which is why we set `metrics=["accuracy"]`.

Training and Evaluating Models

Now the model is ready to be trained. For this we simply need to call its `fit()` method:

C.R. 18 python
```
history = model.fit(X_train, y_train, epochs=30,
                    validation_data=(X_valid, y_valid))
```

We pass it the input features (`X_train`) and the target classes (`y_train`), as well as the number of epochs to train (or else it would default to just 1, which would definitely not be enough to converge to a good solution).

We also pass a validation set which is optional. Keras will measure the loss and the extra metrics on this set at the end of each epoch, which is very useful to see how well the model really performs.

> If the performance on the training set is much better than on the validation set, the model is probably overfitting the training set, or there is a bug, such as a data mismatch between the training set and the validation set.

And that's it! The neural network is trained. At each epoch during training, Keras displays the number of mini-batches processed so far on the left side of the progress bar.

The batch size is 32 by default, and since the training set has 55,000 images, the model goes through 1,719 batches per epoch: 1,718 of size 32, and 1 of size 24.
After the progress bar, you can see the mean training time per sample, and the loss and accuracy (or any other extra metrics you asked for) on both the training set and the validation set and notice that the training loss went down, which is a good sign, and the validation accuracy reached 88.94% after 30 epochs.

That's slightly below the training accuracy, so there is a little bit of overfitting going on, but not a huge amount.
If the training set was very skewed, with some classes being overrepresented and others underrepresented, it would be useful to set the `class_weight` argument when calling the `fit()` method, to give a larger weight to underrepresented classes and a lower weight to overrepresented classes.

These weights would be used by Keras when computing the loss. If you need per-instance weights, set the `sample_weight` argument. If both `class_weight` and `sample_weight` are provided, then Keras multiplies them. Per-instance weights could be useful, for example, if some instances were labeled by experts while others were labeled using a crowdsourcing platform: you might want to give more weight to the former.

You can also provide sample weights (but not class weights) for the validation set by adding them as a third item in the `validation_data` tuple. The `fit()` method returns a History object containing the training parameters (history.params), the list of epochs it went through (history.epoch), and most importantly a dictionary (history.history) containing the loss and extra metrics it measured at the end of each epoch on the training set and on the validation set (if any).

C.R. 19 python
```python
print(history.params)
print(history.epoch)
```

C.R. 20 text
```
{'verbose': 'auto', 'epochs': 30, 'steps': 1719}
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
↪  25, 26, 27, 28, 29]
```

If you use this dictionary to create a Pandas DataFrame and call its `plot()` method, you get the learning curves shown in Fig. 1.11.
You can see that both the training accuracy and the validation accuracy steadily increase during training, while the training loss and the validation loss decrease.

This is good.

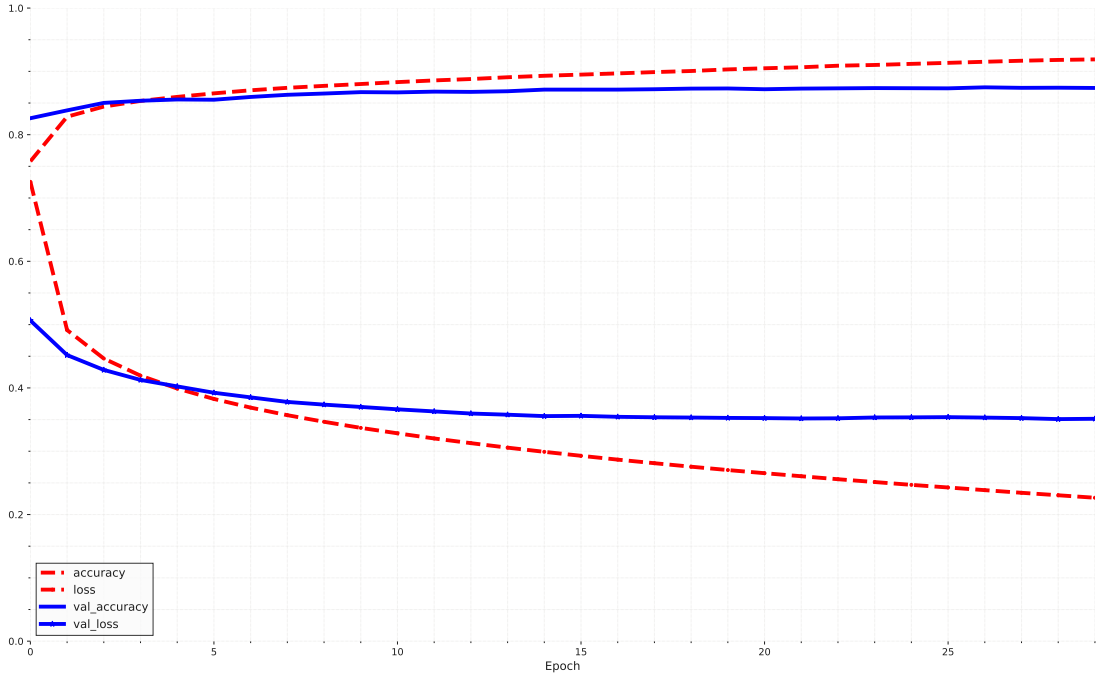

Figure 1.11.: Learning curves: the mean training loss and accuracy measured over each epoch, and the mean validation loss and accuracy measured at the end of each epoch

The validation curves are relatively close to each other at first, but they get further apart over time, which shows that there's a little bit of overfitting. In this particular case, the model looks like it performed better on the validation set than on the training set at the beginning of training, but that's not actually the case.

The validation error is computed at the end of each epoch, while the training error is computed using a running mean during each epoch, so the training curve should be shifted by half an epoch to the left.

If you do that, you will see that the training and validation curves overlap almost perfectly at the beginning of training. The training set performance ends up beating the validation performance, as is generally the case when you train for long enough.

You can tell that the model has not quite converged yet, as the validation loss is still going down, so it would be better to continue training. This is as simple as calling the `fit()` method again, as Keras just continues training where it left off: you should be able to reach about 89.8% validation accuracy, while the training accuracy will continue to rise up to 100%.

> This is not always the case.

If you are not satisfied with the performance of your model, it is a good idea to back and tune the hyperparameters.

1. First check the learning rate (η).

2. If that doesn't help, try another optimizer, and always retune the learning rate after changing any hyperparameter,
3. If the performance is still not great, try tuning model hyperparameters such as the number of layers, the number of neurons per layer, and the types of activation functions to use for each hidden layer.

   You can also try tuning other hyperparameters, such as the batch size (it can be set in the `fit()` method using the `batch_size` argument, which defaults to 32).

Once you are satisfied with your model's validation accuracy, you should evaluate it on the test set to estimate the generalization error before you deploy the model to production. You can easily do this using the `evaluate()` method.

> It also supports several other arguments, such as `batch_size` and `sample_weight`.

It is common to get slightly lower performance on the test set than on the validation set, as hyperparameters are **tuned on the validation set**, not the test set. However, in this example, we did not do any hyperparameter tuning, so the lower accuracy is just bad luck.

> Resist the temptation to tweak the hyperparameters on the test set, or else your estimate of the generalization error will be too optimistic.

Using Model to Make Predictions

It is time to use the model's `predict()` method to make predictions on new instances. As we don't have actual new instances, we'll just use the first three (3) instances of the test set:

C.R. 21 python

```python
X_new = X_test[:3]
y_proba = model.predict(X_new)
print(y_proba.round(2))
```

C.R. 22 text

```
[[0.   0.   0.   0.   0.   0.12 0.   0.01 0.   0.87]
 [0.   0.   1.   0.   0.   0.   0.   0.   0.   0.  ]
 [0.   1.   0.   0.   0.   0.   0.   0.   0.   0.  ]]
```

For each instance the model estimates one probability per class, from class 0 to class 9. This is similar to the output of the `predict_proba()` method in `sklearn` classifiers.

For example, for the first image it estimates that the probability of class 9 (ankle boot) is 87%, the probability of class 7 (sneaker) is 1%, the probability of class 5 (sandal) is 12%, and the probabilities of the other classes are negligible.

In other words, it is highly confident that the first image is footwear, most likely ankle boots but possibly sneakers or sandals. If you only care about the class with the highest estimated probability (even if that probability is quite low), then you can use the `argmax()` method to get the highest probability class index for each instance:

C.R. 23 python
```python
y_pred = y_proba.argmax(axis=-1)
print(y_pred)
```

C.R. 24 text
```
[9 2 1]
```

Here, the classifier actually classified all three images correctly, where these images are shown in Fig. 1.12.

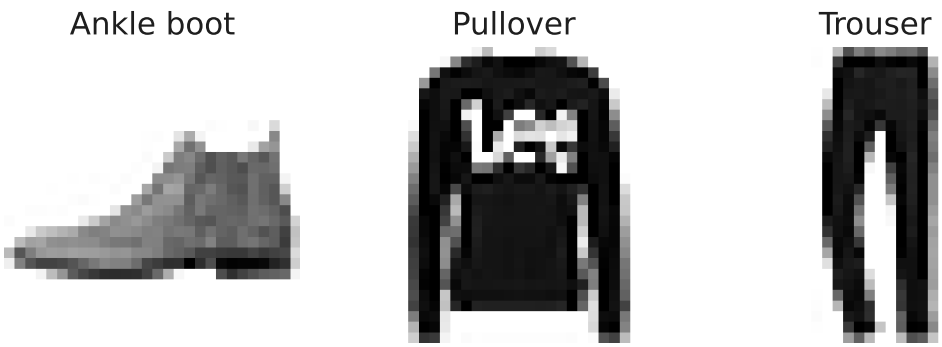

Figure 1.12.: Correctly classified Fashion MNIST images.

1.3.3 Building a Regression MLP Using the Sequential API

Instead of classifying categories, lets try to estimate a value. For this application, we need a different dataset. Let's switch back to the California housing problem and tackle it using the same MLP as earlier, with 3 hidden layers composed of 50 neurons each, but this time building it with `tf.keras`.

Using the sequential API to build, train, evaluate, and use a regression MLP is quite similar to what we did for classification. The main differences in the following code example are the fact that the output layer has a single neuron (since we only want to predict a single value) and it uses no activation function, the loss function is the mean squared error, the metric is the RMSE, and we're using an Adam optimizer like `sklearn`s `MLPRegressor` did.

In addition, in this example we don't need a Flatten layer, and instead we're using a Normalization layer as the first layer: it does the same thing as `sklearn`s `StandardScaler`, but it must be fitted to the training data using its `adapt()` method before you call the model's `fit()` method.

Let's look at the code:

C.R. 25 python
```python
housing = fetch_california_housing()
X_train_full, X_test, y_train_full, y_test = train_test_split(
    housing.data, housing.target, random_state=42)
X_train, X_valid, y_train, y_valid = train_test_split(
    X_train_full, y_train_full, random_state=42)
```

C.R. 26
python
```python
tf.random.set_seed(42)
norm_layer = tf.keras.layers.Normalization(input_shape=X_train.shape[1:])
model = tf.keras.Sequential([
    norm_layer,
    tf.keras.layers.Dense(50, activation="relu"),
    tf.keras.layers.Dense(50, activation="relu"),
    tf.keras.layers.Dense(50, activation="relu"),
    tf.keras.layers.Dense(1)
])
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-3)
model.compile(loss="mse", optimizer=optimizer, metrics=["RootMeanSquaredError"])
norm_layer.adapt(X_train)
history = model.fit(X_train, y_train, epochs=20,
                    validation_data=(X_valid, y_valid))
mse_test, rmse_test = model.evaluate(X_test, y_test)
X_new = X_test[:3]
y_pred = model.predict(X_new)
```

As you can see, the sequential API is quite clean and straightforward. However, although Sequential models are extremely common, it is sometimes useful to build neural networks with more complex topologies, or with multiple inputs or outputs. For this purpose, Keras offers the functional API.

Chapter 2

Computer Vision using Convolutional Neural Networks

Table of Contents

2.1 Introduction

It wasn't until recently computers were able to reliably perform seemingly easy tasks such as detecting a cat[1] in a picture or recognising spoken words.

Why are these tasks so effortless to us humans?

[1] As it is known, internet was invented for sharing cat pictures, therefore their detection is paramount.

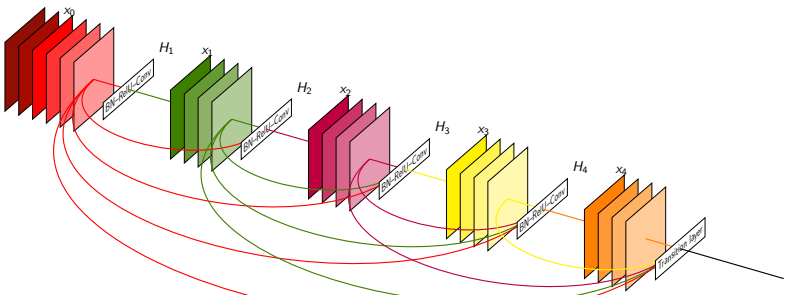

Figure 2.1.: A standard CNN architecture. Don't worry if it looks too confusing at the moment as by the end of this chapter we will have the knowledge to understand and build your very own CNN.

The answer lies in the fact that perception largely takes place outside the realm of our control, within specialized visual, auditory, and other sensory modules in our brains. By the time sensory information reaches our consciousness, it is already imbued with high-level features. For example:

> When we look at a picture of a cute puppy, we cannot choose not to see the puppy, not to notice its cuteness. Nor can we explain how we recognize a cute puppy; it's just obvious to you.

Therefore, we cannot trust our subjective experience.

Perception is **NOT** a trivial task, and to understand it we must look at how our sensory modules work. Convolutional Neural Networks (CNN)s emerged from the study of the brain's visual cortex, and they have been used in computer image recognition since the 1980s [43].

Over the last 10 years, thanks to the increase in computational power, the amount of available training data, and a much better methods developed for training deep nets, CNNs have managed to achieve impressive performance on some complex visual tasks. Some examples of their application include:

- **Search services:** Such as connecting users in the web to the software they need to find [20, 26],
- **Self-driving cars:** Such as detecting the colours on a traffic light [31], or detecting the road lines during driving [30],
- **Automatic video classification**: i.e., detecting videos and categorising based on content [42],
- **Object Detection:** Such as detecting objects in an image [45].

In addition, CNNs are not restricted to visual perception:

> They are also successful at many other tasks, such as voice recognition and natural language processing.

However, as our topic is **Image Processing**, we will focus on its visual applications. In this chapter we will explore where CNNs came from, what their building blocks look like, and how to implement them using `tf.keras`. Then we will discuss some of the best CNN architectures, as well as other visual tasks, including object detection[2] and semantic segmentation..

[2] classifying multiple objects in an image and placing bounding boxes around them

2.2 Visual Cortex Architecture

David H. Hubel and *Torsten Wiesel* performed a series of experiments on cats in 1958 and 1959 (and a few years later on monkeys), giving crucial insights into the structure of the visual cortex[3] [21]. What is important to us is, they showed that many neurons in the visual cortex have a small local receptive field, meaning they react only to visual stimuli located in a limited region of the visual field. A diagram showcasing this phenomenon can be seen in **Fig.** 2.2 , in which the local receptive fields of five neurons are represented by dashed circles. The receptive fields of different neurons may overlap, and together they tile the whole visual field.

[3] the authors received the Nobel Prize in Physiology or Medicine in 1981 for their work

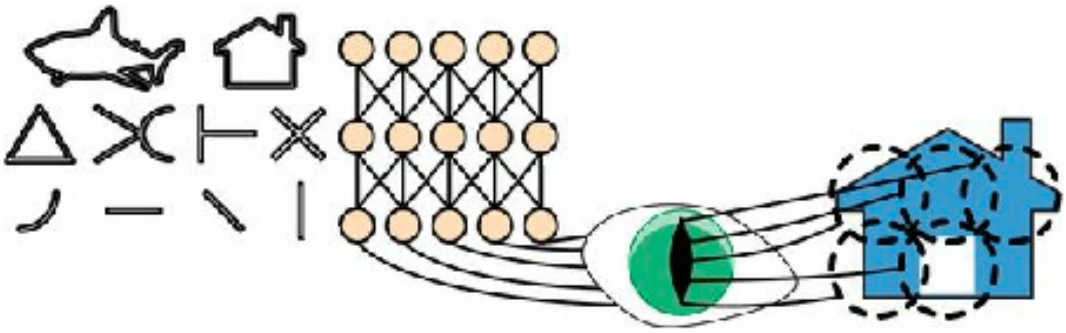

Figure 2.2.: Biological neurons in the visual cortex respond to specific patterns in small regions of the visual field called receptive fields; as the visual signal makes its way through consecutive brain modules, neurons respond to more complex patterns in larger receptive fields

In addition to the previous revelation of how neurons work, they showed some neurons react **ONLY** to images of horizontal lines, while others react **ONLY** to lines with different orientations[4]. They also noticed that some neurons have larger receptive fields, and they react to more complex patterns that are combinations of the lower-level patterns. These observations led to the idea that the higher-level neurons are based on the outputs of neighboring lower-level neurons. (in **Fig.** 2.2 , observe that each neuron is connected only to nearby neurons from the previous layer).

[4] two neurons may have the same receptive field but react to different line orientations

This powerful architecture is able to detect all sorts of complex patterns in any area of the visual field. These studies of the visual cortex inspired the neocognitron [9] , introduced in 1980, which gradually evolved into what we now call CNN.

An important milestone was a 1998 paper by *Yann LeCun et.al.* which introduced the famous LeNet-5 architecture, which became widely used by banks to recognize handwritten digits on checks [25]. This architecture has some building blocks were are familiar with:

- Fully connected layers,
- Sigmoid activation functions[5],

but it also introduces two new building blocks:

- convolutional layers
- pooling layers.

Which will, of course, will be our focus of attention this chapter.

[5] A function allowing non-linear properties for the neural network.

Limits of DNN

Why not simply use a DNN with fully connected layers for image recognition tasks? Why do we need a new method ?

Unfortunately, although this works fine for small images (e.g., MNIST), it breaks down for larger images due to the huge number of parameters it requires.
For example, a 100-by-100 pixel image has 10,000 pixels, and if the first layer has just 1,000 neurons (which already severely restricts the amount of information transmitted to the next layer), this means a total of 10 million connections. And that's just the first layer.

CNNs solve this problem using partially connected layers and weight sharing.

2.3 Convolutional Layers

The most important building block of a CNN is the convolutional layer:

> Neurons in the first convolutional layer are **NOT** connected to every single pixel in the input image[6].

[6] This approach is going against the previously discussed ANNs and DNNs

The neurons are only connected to pixels in their receptive fields which its graphical representation can be seen in **Fig.** 2.3 .

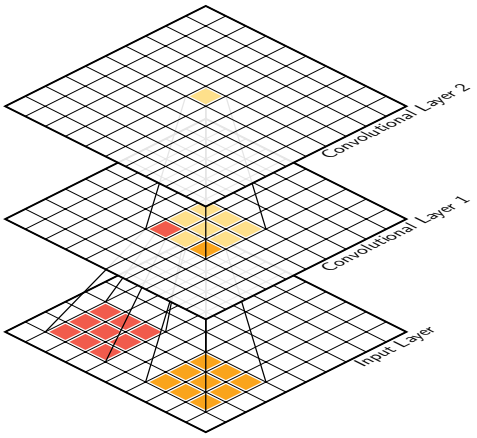

Figure 2.3.: CNN layers with rectangular local receptive fields.

In a CNN, each layer is represented in 2D, which makes it easier to match neurons with their corresponding inputs as our images are also 2D.

In turn, each neuron in the second convolutional layer is connected only to neurons located within a small rectangle in the first layer. This architecture allows the network to concentrate on small low-level features in the first hidden layer, then assemble them into larger higher-level features in the next hidden layer, and so on. This hierarchical structure is common in real-world images, which is one of the reasons why CNNs work so well for image recognition.

A neuron located in row i, column j of a given layer is connected to the outputs of the neurons in the previous layer located in rows i to $i + f_h - 1$, columns j to $j + f_w - 1$, where f_h and f_w are the height and width of the receptive field which can be observed in **Fig. ??** .

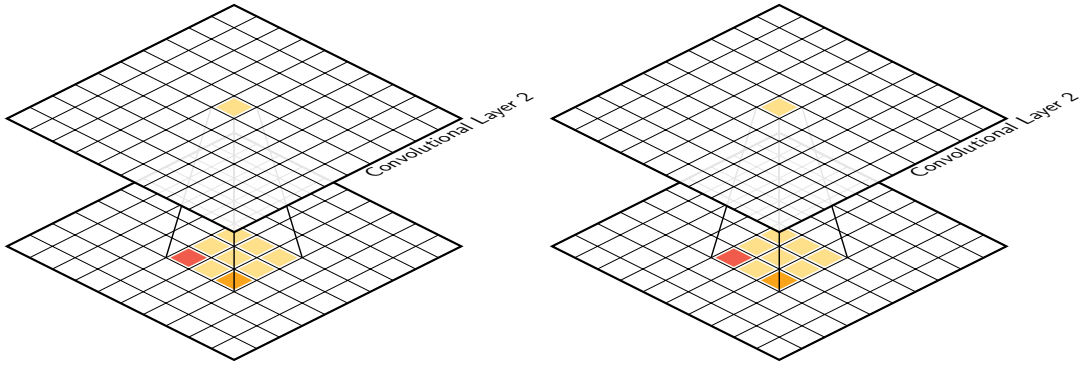

Figure 2.4.

> For a layer to have the same height and width as the previous layer, it is common to add zeros around the inputs, which is called *zero padding*.

It is also possible to connect a large input layer to a much smaller layer by spacing out the receptive fields, shown in **Fig. ??** .

This significantly decreases the model's computational complexity. The horizontal or vertical step size from one receptive field to the next is called the **stride**. In **Fig. ??** , a 5-by-7 input layer (plus zero padding) is connected to a 3-by-4 layer, using 3-by-3 receptive fields and a stride of 2[7]. A neuron located in row i, column j in the upper layer is connected to the outputs of the neurons in the previous layer located in rows i×sh toi×sh +fh –1,columnsj×sw to j × sw + fw – 1, where sh and sw are the vertical and horizontal strides.

[7] in this example the stride is the same in both directions, but it does not have to be so

2.3.1 Filters

A neuron's weights can be represented as a small image the size of the receptive field. For example, **Fig.** 2.5 shows two (2) possible sets of weights, called **filters**[8].

[8] The terms convolution kernels, or kernels are also used

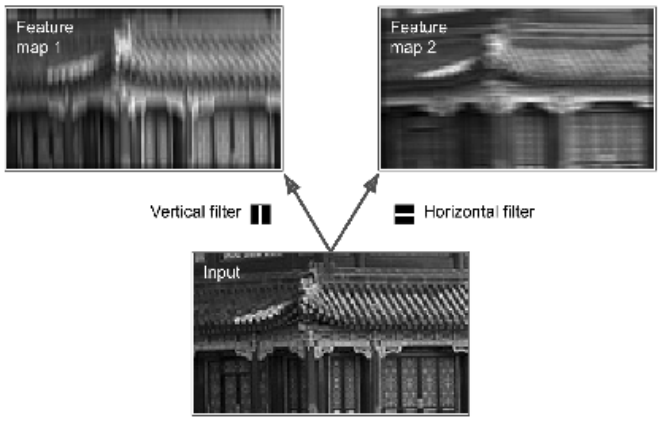

Figure 2.5.: An example image showcasing the effect of applying filters to get feature maps.

The first one is represented as a black square with a vertical white line in the middle.

> It's a 7-by-7 matrix full of 0s except for the central column, which is full of 1s.

Neurons using these weights will ignore everything in their receptive field except for the central vertical line[9]. The second filter is a black square with a horizontal white line in the middle. Neurons using these weights will ignore everything in their receptive field except for the central horizontal line.

[9] since all inputs will be multiplied by 0, except for the ones in the central vertical line

If all neurons in a layer use the same vertical line filter (and the same bias term), and we feed the network the input image shown in **Fig.** 2.5 (the bottom image), the layer will output the top-left image. Notice that the vertical white lines get enhanced while the rest gets blurred. Similarly, the upper-right image is what we get if all neurons use the same horizontal line filter. Notice the horizontal white lines get enhanced while the rest is blurred out. Therefore, a layer full of neurons using the same filter outputs a feature map, which highlights the areas in an image that activate the filter the most.

> We won't have to define the filters manually: instead, during training the convolutional layer will automatically learn the most useful filters for its task, and the layers above will learn to combine them into more complex patterns.

2.3.2 Stacking Multiple Feature Maps

Up to now, for simplicity, We represented the output of each convolutional layer as a 2D layer, but in reality a convolutional layer has multiple filters[10] and outputs one feature map per filter, so it is more accurately represented in 3D, which can be seen in **Fig.** 2.6 .

[10] The number of filters is left up to the programmer

It has one neuron per pixel in each feature map, and all neurons within a given feature map share the same parameters (i.e., the same kernel and bias term). Neurons in different feature maps use different parameters. A neuron's receptive field is the same as described earlier, but it extends across all the feature maps of the previous layer. In short, a convolutional layer simultaneously applies multiple trainable filters to its inputs, making it capable of detecting multiple features anywhere in its inputs.

> The fact that all neurons in a feature map share the same parameters dramatically reduces the number of parameters in the model. Once the CNN has learned to recognize a pattern in one location, it can recognize it in any other location. In contrast, once a fully connected neural network has learned to recognize a pattern in one location, it can only recognize it in that particular location.

Input images are also composed of multiple sublayers: one per color channel. As we already now, there are typically three: red, green, and blue (RGB). Grayscale images have just one channel, but some images may have many more—for example, satellite images that capture extra light frequencies

Specifically, a neuron located in row i, column j of the feature map k in a given convolutional layer l is connected to the outputs of the neurons in the previous layer $l-1$, located in rows $i \times s_h$ to $i \times s_h + f_h - 1$ and columns $j \times s_w$ to $j \times s_w + f_w - 1$,across all feature maps(in layer $l-1$).

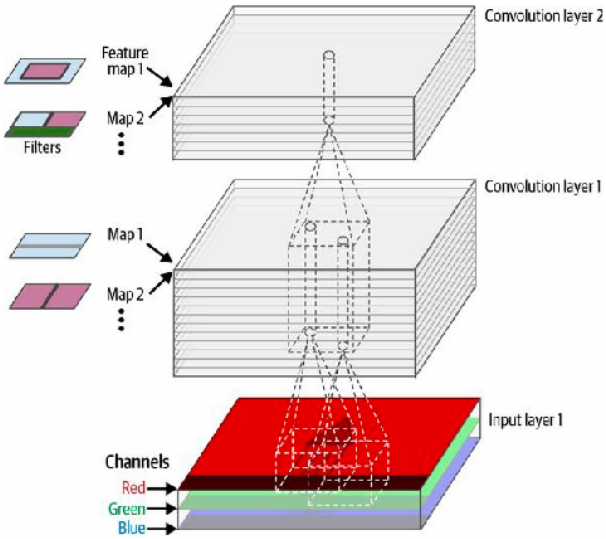

Figure 2.6.

> Within a layer, all neurons located in the same row i and column j but in different feature maps are connected to the outputs of the exact same neurons in the previous layer.

This definition can be summarised in one big mathematical equation: it shows how to compute the output of a given neuron in a convolutional layer.
Let's try to understand the variables:

- $z_{i,j,k}$ is the output of the neuron located in row i, column j in feature map k of the convolutional layer (layer l).
- s_h, s_w are the vertical and horizontal strides, f_h, f_w are the height and width of the receptive field, and f'_n is the number of feature maps in the previous layer ($l-1$).
- $x_{i',j',k'}$ is the output of the neuron located in layer $l-1$, row i', column j', feature map k', or channel k' if the previous layer is the input layer.
- b_k is the bias term for feature map k (in layer l). Think of it as a knob that tweaks the overall brightness of the feature map k.
- $w_{u,v,k',k}$ is the connection weight between any neuron in feature map k of the layer l and its input located at row u, column v (relative to the neuron's receptive field), and feature map k'.

2.3.3 Implementing Convolutional Layers with Keras

As with every ML application, we load and preprocess a couple of sample images, using `sklearn`'s `load_sample_image()` function and Keras's CenterCrop and Rescaling layers:

C.R.1 python
```python
from sklearn.datasets import load_sample_images
import tensorflow as tf

```

C.R. 2
```python
images = load_sample_images()["images"]
images = tf.keras.layers.CenterCrop(height=70, width=120)(images)
images = tf.keras.layers.Rescaling(scale=1 / 255)(images)
```

Let's look at the shape of the images tensor[11]:

[11] A tensor is a generalization of vectors and matrices to potentially higher dimensions.

C.R. 3
```python
print(images.shape)
```

```text
(2, 70, 120, 3)
```

It's a 4D tensor. Let's see why this is a 4D matrix. There are two (2) sample images, which explains the first dimension. Then each image is 70-by-120, as that's the size we specified when creating the CenterCrop layer[12]. This explains the second and third dimensions. And lastly, each pixel holds one value per colour channel, and there are three (3) of them:

[12] the original images were 427-by-640

Red, Green, and Blue.

Now let's create a 2D convolutional layer and feed it these images to see what comes out. For this, Keras provides a Convolution2D layer, alias `Conv2D`. Under the hood, this layer relies on TensorFlow's `tf.nn.conv2d()` operation.

Let's create a convolutional layer with 32 filters, each of size 7-by-7 (using `kernel_size=7`, which is equivalent to using `kernel_size=(7 , 7)`), and apply this layer to our small batch of two (2) images:

C.R. 4
```python
tf.random.set_seed(42)  # extra code - ensures reproducibility
conv_layer = tf.keras.layers.Conv2D(filters=32, kernel_size=7)
fmaps = conv_layer(images)
```

Now let's look at the output's shape:

C.R. 5
```python
print(fmaps.shape)
```

```text
(2, 64, 114, 32)
```

The output shape is similar to the input shape, with two (2) main differences.

1. There are 32 channels instead of 3. This is because we set `filters=32`, so we get 32 output feature maps: instead of the intensity of red, green, and blue at each location, we now have the intensity of each feature at each location.

2. The height and width have both shrunk by 6 pixels. This is due to the fact that the Conv2D layer does **NOT** use any zero-padding by default, which means that we lose a few pixels on the sides of the output feature maps, depending on the size of the filters. In this case, since the kernel size is 7, we lose 6 pixels horizontally and 6 pixels vertically (i.e., 3 pixels on each side).

   If instead we set `padding="same"`, then the inputs are padded with enough zeros on all sides to ensure that the output feature maps end up with the same size as the inputs (hence the name of this option):

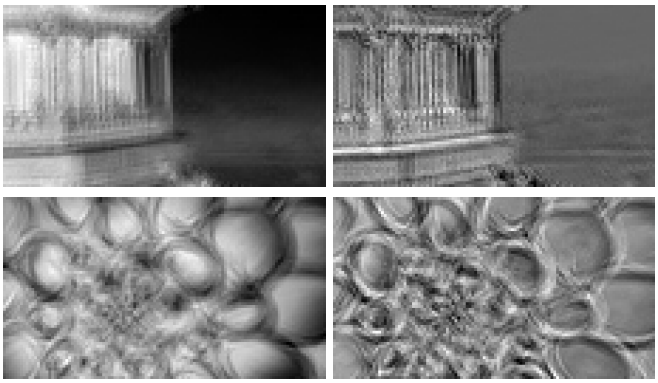

Figure 2.7.

These two padding options are illustrated in **Fig.** 2.8 . For simplicity, only the horizontal dimension is shown here, but of course the same logic applies to the vertical dimension as well.

If the stride is greater than 1 (in any direction), then the output size will not be equal to the input size, even if `padding="same"`. For example, if we set `strides=2` (or equivalently `strides=(2, 2)`), then the output feature maps will be 35-by-60: halved both vertically and horizontally.

Fig. 2.8 shows what happens when **strides=2**, with both padding options.

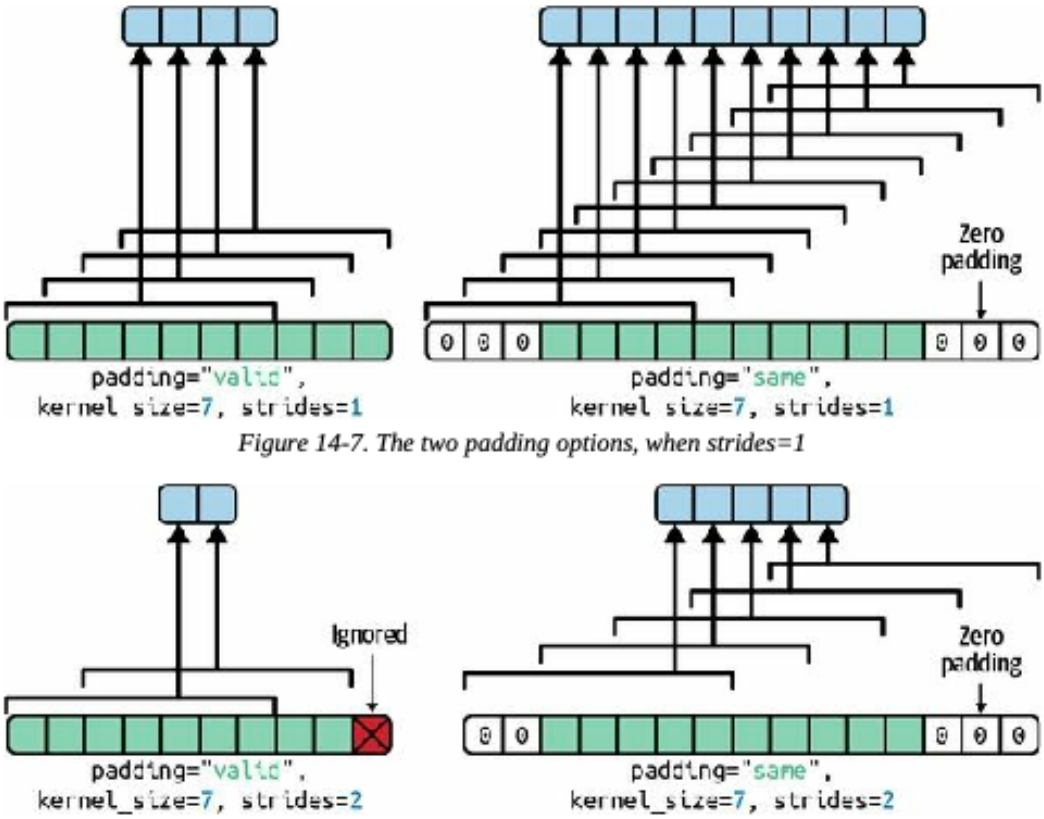

Figure 2.8.: Showcasing different padding options.

If we are curious, this is how the output size is computed:

- With `padding="valid"`, if the width of the input is ih – fh + sh) / sh, rounded down. Recall that fh is the kernel width, and sh is the horizontal stride. Any remainder in the division corresponds to ignored columns on the right side of the input image. The same logic can be used to compute the output height, and any ignored rows at the bottom of the image.
- With `padding="same"`, the output width is equal to ih / sh, rounded up. To make this possible, the appropriate number of zero columns are padded to the left and right of the input image (an equal number if possible, or just one more on the right side). Assuming the output width is ow, then the number of padded zero columns is (ow – 1) × sh + fh – ih. Again, the same logic can be used to compute the output height and the number of padded rows.

Now let's look at the layer's weights (which were noted wu, v, k', k and bk in Equation 14-1). Just like a Dense layer, a Conv2D layer holds all the layer's weights, including the kernels and biases. The kernels are initialized randomly, while the biases are initialized to zero. These weights are accessible as TF variables via the weights attribute, or as NumPy arrays via the `get_weights()` method:

The kernels array is 4D, and its shape is [`kernel_height`, `kernel_width`, `input_channels`, `output_channels`]. The biases array is 1D, with shape [`output_channels`]. The number of output channels is equal to the number of output feature maps, which is also equal to the number of filters. Most importantly, note that the height and width of the input images do not appear in the kernel's shape: this is because all the neurons in the output feature maps share the same weights, as explained earlier. This means that we can feed images of any size to this layer, as long as they are at least as large as the kernels, and if they have the right number of channels (three in this case).

Lastly, we will generally want to specify an activation function (such as ReLU) when creating a Conv2D layer, and also specify the corresponding kernel initializer. This is for the same reason as for Dense layers: a convolutional layer performs a linear operation, so if we stacked multiple convolutional layers without any activation functions they would all be equivalent to a single convolutional layer, and they wouldn't be able to learn anything really complex.

As we can see, convolutional layers have quite a few hyperparameters: filters, `kernel_size`, padding, strides, activation, `kernel_initializer`, etc. As always, we can use cross-validation to find the right hyperparameter values, but this is very time-consuming. We will discuss common CNN architectures later in this chapter, to give you some idea of which hyperparameter values work best in practice.

2.3.4 Memory Requirements

Another challenge with CNNs is that the convolutional layers require a huge amount of RAM. This is especially true during training, because the reverse pass of back-propagation requires all the intermediate values computed during the forward pass.

As an example, consider a convolutional layer with 200 5-by-5 filters, with stride 1 and `"same"` padding. If the input is a 150-by-100 RGB image, then the number of parameters is (5-by-5-by-3+1) × 200 = 15,200 [13], which is fairly small compared to a fully connected layer. However, each of the 200 feature

[13] the + 1 corresponds to the bias terms

maps contains 150 × 100 neurons, and each of these neurons needs to compute a weighted sum of its 5-by-5-by-3 = 75 inputs: that's a total of 225 million float multiplications.

Not as bad as a fully connected layer, but still quite computationally intensive. Moreover, if the feature maps are represented using 32-bit floats, then the convolutional layer's output will occupy $200 \times 150 \times 100 \times 32 = 96$ million bits (12 MB) of RAM. 8 And that's just for one instance—if a training batch contains 100 instances, then this layer will use up 1.2 GB of RAM.

During inference (i.e., when making a prediction for a new instance) the RAM occupied by one layer can be released as soon as the next layer has been computed, so we only need as much RAM as required by two consecutive layers. But during training everything computed during the forward pass needs to be preserved for the reverse pass, so the amount of RAM needed is (at least) the total amount of RAM required by all layers.

Now let's look at the second common building block of CNNs: the pooling layer.

2.4 Pooling Layer

The goal of a pooling layer is to subsample (i.e., shrink) the input image to reduce the computational load, the memory usage, and the number of parameters [14].

[14] This limits the risk of overfitting

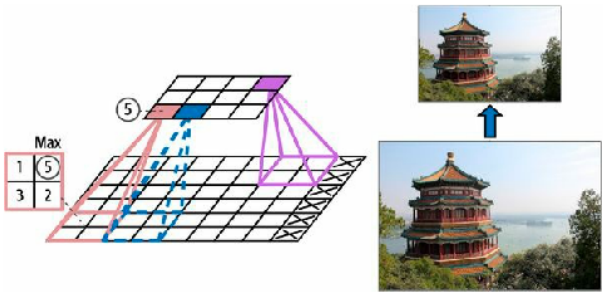

Figure 2.9.

Just like in convolutional layers, each neuron in a pooling layer is connected to the outputs of a limited number of neurons in the previous layer, located within a small rectangular receptive field. We must define its size, the stride, and the padding type, just like before.

However, a pooling neuron has **NO** weights. All it does is aggregate the inputs using an aggregation function such as the max or mean.

Fig. 2.9 shows a **max pooling layer**, which is the most common type of pooling layer. In this example, we use a 2-by-2 pooling kernel, with a stride of 2 and no padding. Only the max input value in each receptive field makes it to the next layer, while the other inputs are dropped. For example, in the lower-left receptive field in **Fig.** 2.9 , the input values are 1, 5, 3, 2, so only the max value, 5, is propagated to the next layer. Because of the stride of 2, the output image has half the height and half the width of the input image (rounded down since we use no padding).

A pooling layer typically works on every input channel independently, so the output depth (i.e., the number of channels) is the same as the input depth.

Other than reducing computations, memory usage, and the number of parameters, a max pooling layer also introduces some level of invariance[15] to small translations, as shown in **Fig.** 2.10 .

[15] In the context of computer vision, it means that we can recognize an object as an object, even when its appearance varies in some way.

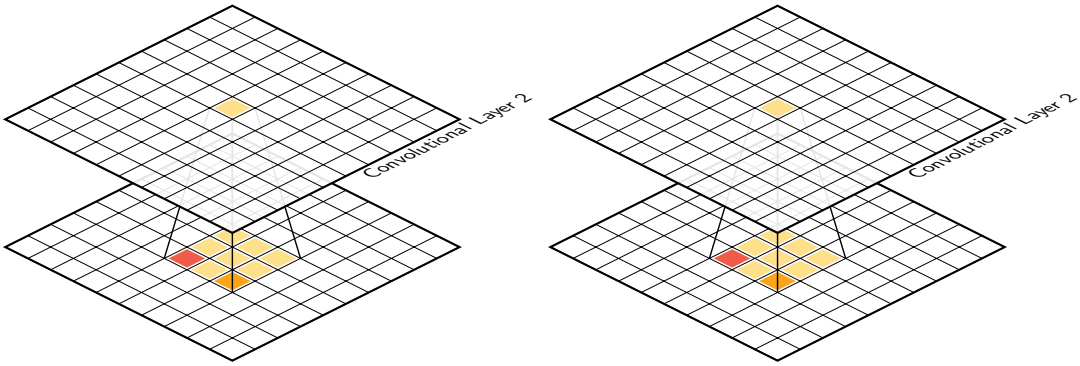

Figure 2.10.: As can be seen from the image above, max-pooling allow a certain level of invariance when the object moves in small increments. This is an important property as it is favourable when small changes in movement don't affect the overall recognition of the image.

Here we assume that the bright pixels have a lower value than dark pixels, and we consider three images (A, B, C) going through a max pooling layer with a 2-by-2 kernel and stride 2. Images B and C are the same as image A, but shifted by one and two pixels to the right. As we can see, the outputs of the max pooling layer for images A and B are identical. This is what translation invariance means. For image C, the output is different: it is shifted one pixel to the right (but there is still 50% invariance). By inserting a max pooling layer every few layers in a CNN, it is possible to get some level of translation invariance at a larger scale.

Moreover, max pooling offers a small amount of rotational invariance and a slight scale invariance. Such invariance (even if it is limited) can be useful in cases where the prediction should not depend on these details, such as in classification tasks.

However, max pooling has some downsides too. It's obviously very destructive: even with a tiny 2 × 2 kernel and a stride of 2, the output will be two times smaller in both directions (so its area will be four times smaller), simply dropping 75% of the input values. And in some applications, invariance is not desirable. Take semantic segmentation [16].

[16] The task of classifying each pixel in an image according to the object that pixel belongs to

For this case, if the input image is translated by one pixel to the right, the output should also be translated by one pixel to the right. The goal in this case is equivariance, not invariance: a small change to the inputs should lead to a coresponding small change in the output.

2.5 Implementing Pooling Layers with Keras

The following code creates a MaxPooling2D layer, alias **MaxPool2D**, using a 2-by-2 kernel. The strides default to the kernel size, so this layer uses a stride of 2 (horizontally and vertically). By default, it uses **"valid"** padding (i.e., no padding at all):

C.R. 6 python
```
max_pool = tf.keras.layers.MaxPool2D(pool_size=2)
```

To create an average pooling layer, just use AveragePooling2D, alias **AvgPool2D**, instead of MaxPool2D. As we might expect, it works exactly like a max pooling layer, except it computes the mean rather than the max. Average pooling layers used to be very popular, but people mostly use max pooling layers now, as they generally perform better. This may seem surprising, since computing the mean generally loses less information than computing the max. But on the other hand, max pooling preserves only the strongest features, getting rid of all the meaningless ones, so the next layers get a cleaner signal to work with. Moreover, max pooling offers stronger translation invariance than average pooling, and it requires slightly less compute.

Note that max pooling and average pooling can be performed along the depth dimension instead of the spatial dimensions, although it's not as common. This can allow the CNN to learn to be invariant to various features. For example, it could learn multiple filters, each detecting a different rotation of the same pattern (such as handwritten digits seen in **Fig.** 2.11), and the depthwise max pooling layer would ensure that the output is the same regardless of the rotation. The CNN could similarly learn to be invariant to anything: thickness, brightness, skew, color, and so on.

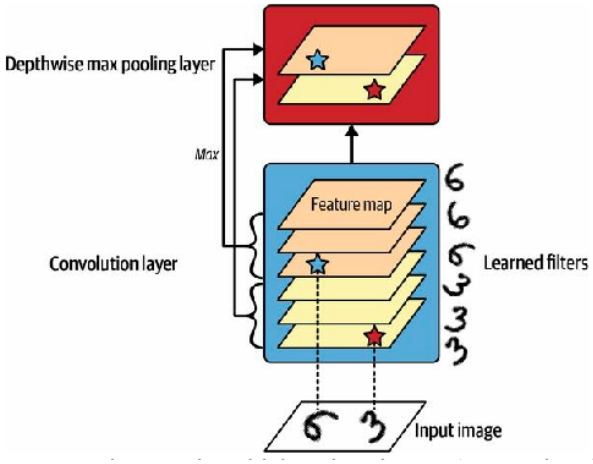

Figure 2.11.: Depthwise max pooling can help the CNN learn to be invariant (to rotation in this case).

Keras does not include a depthwise max pooling layer, but it's not too difficult to implement a custom layer for that:

C.R. 7 python
```
class DepthPool(tf.keras.layers.Layer):
    def __init__(self, pool_size=2, **kwargs):
        super().__init__(**kwargs)
        self.pool_size = pool_size

    def call(self, inputs):
        shape = tf.shape(inputs)  # shape[-1] is the number of channels
        groups = shape[-1] // self.pool_size  # number of channel groups
        new_shape = tf.concat([shape[:-1], [groups, self.pool_size]], axis=0)
        return tf.reduce_max(tf.reshape(inputs, new_shape), axis=-1)
```

This layer reshapes its inputs to split the channels into groups of the desired size (`pool_size`), then it uses `tf.reduce_max()` to compute the max of each group. This implementation assumes that the stride is equal to the pool size, which is generally what we want. Alternatively, we could use TensorFlow's `tf.nn.max_pool()` operation, and wrap in a Lambda layer to use it inside a Keras model, but sadly this op does not implement depthwise pooling for the GPU, only for the CPU.

One last type of pooling layer that we will often see in modern architectures is the global average pooling layer. It works very differently: all it does is compute the mean of each entire feature map (it's like an average pooling layer using a pooling kernel with the same spatial dimensions as the inputs). This means that it just outputs a single number per feature map and per instance. Although this is of course extremely destructive (most of the information in the feature map is lost), it can be useful just before the output layer, as we will see later in this chapter. To create such a layer, simply use the GlobalAveragePooling2D class, alias GlobalAvgPool2D:

2.6 CNN Architectures

Typical CNN architectures stack a few convolutional layers (each one generally followed by a ReLU layer), then a pooling layer, then another few convolutional layers (+ReLU), then another pooling layer, and so on. The image gets smaller and smaller as it progresses through the network, but it also typically gets deeper and deeper (i.e., with more feature maps), thanks to the convolutional layers (see **Fig. ??**). At the top of the stack, a regular feedforward neural network is added, composed of a few fully connected layers (+ReLUs), and the final layer outputs the prediction (e.g., a softmax layer that outputs estimated class probabilities).

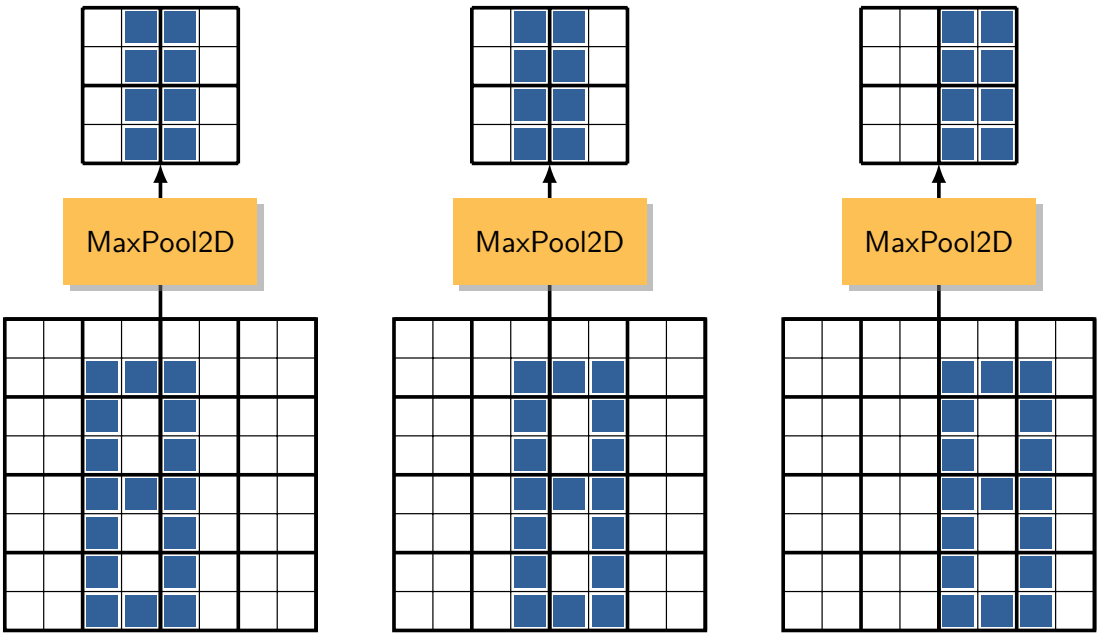

Figure 2.12.: An example structure of a CNN network

> A common mistake is to use convolution kernels that are too large. For example, instead of using a convolutional layer with a 5 × 5 kernel, stack two layers with 3 × 3 kernels: it will use fewer parameters and require fewer computations, and it will usually perform better. One exception is for the first convolutional layer: it can typically have a large kernel (e.g., 5 × 5), usually with a stride of 2 or more. This will reduce the spatial dimension of the image without losing too much information, and since the input image only has three channels in general, it will not be too costly.

Here is how we can implement a basic CNN to tackle the Fashion MNIST dataset.
Let's first download and allocated the traning/testing.

C.R. 8 python

```python
import numpy as np

mnist = tf.keras.datasets.fashion_mnist.load_data()
(X_train_full, y_train_full), (X_test, y_test) = mnist
X_train_full = np.expand_dims(X_train_full, axis=-1).astype(np.float32) / 255
X_test = np.expand_dims(X_test.astype(np.float32), axis=-1) / 255
X_train, X_valid = X_train_full[:-5000], X_train_full[-5000:]
y_train, y_valid = y_train_full[:-5000], y_train_full[-5000:]
```

C.R. 9 python

```python
from functools import partial

tf.random.set_seed(42)  # extra code - ensures reproducibility
DefaultConv2D = partial(tf.keras.layers.Conv2D, kernel_size=3, padding="same",
                        activation="relu", kernel_initializer="he_normal")
model = tf.keras.Sequential([
    DefaultConv2D(filters=64, kernel_size=7, input_shape=[28, 28, 1]),
    tf.keras.layers.MaxPool2D(),
    DefaultConv2D(filters=128),
    DefaultConv2D(filters=128),
    tf.keras.layers.MaxPool2D(),
    DefaultConv2D(filters=256),
    DefaultConv2D(filters=256),
    tf.keras.layers.MaxPool2D(),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(units=128, activation="relu",
                          kernel_initializer="he_normal"),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(units=64, activation="relu",
                          kernel_initializer="he_normal"),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(units=10, activation="softmax")
])
```

Let's go through this code:

1. We use the `functools.partial()` function to define DefaultConv2D, which acts just like Conv2D but with different default arguments: a small kernel size of 3, `"same"` padding, the ReLU activation function, and its corresponding *He initializer*.

2. We create the Sequential model. Its first layer is a DefaultConv2D with 64 fairly large filters (7-by-7). It uses the default stride of 1 because the input images are not very large. It also sets `input_shape=[28, 28, 1]`,

as the images are 28-by-28 pixels, with a single color channel (i.e., grayscale).

3. When we load the Fashion MNIST dataset, we need to make sure each image has this shape. Therefore we may require to use `np.reshape()` or `np.expanddims()` to add the channels dimension or we could use a Reshape layer as the first layer in the model.

4. We then add a max pooling layer that uses the default pool size of 2, so it divides each spatial dimension by a factor of 2.

5. We repeat the same structure twice: two convolutional layers followed by a max pooling layer. For larger images, we could repeat this structure several more times. The number of repetitions is a hyperparameter we can tune

> Observe the number of filters doubles as we climb up the CNN toward the output layer (it is initially 64, then 128, then 256): it makes sense for it to grow, since the number of low-level features is often fairly low (e.g., small circles, horizontal lines), but there are many different ways to combine them into higher-level features. It is a common practice to double the number of filters after each pooling layer: since a pooling layer divides each spatial dimension by a factor of 2, we can afford to double the number of feature maps in the next layer without fear of exploding the number of parameters, memory usage, or computational load.

6. Next is the fully connected network, composed of two hidden dense layers and a dense output layer. Since it's a classification task with 10 classes, the output layer has 10 units, and it uses the softmax activation function. Note that we must flatten the inputs just before the first dense layer, since it expects a 1D array of features for each instance. We also add two dropout layers, with a dropout rate of 50% each, to reduce overfitting.

If we compile this model using the `"sparse_categorical_crossentropy"` loss and we fit the model to the Fashion MNIST training set, it should reach over 92% accuracy on the test set.

C.R. 10 python

```
model.compile(loss="sparse_categorical_crossentropy", optimizer="nadam",
              metrics=["accuracy"])
history = model.fit(X_train, y_train, epochs=10,
                    validation_data=(X_valid, y_valid))
score = model.evaluate(X_test, y_test)
X_new = X_test[:10]  # pretend we have new images
y_pred = model.predict(X_new)
```

Over the years, variants of this fundamental architecture have been developed, leading to amazing advances in the field. A good measure of this progress is the error rate in competitions such as the ILSVRC ImageNet challenge. In this competition, the top-five error rate for image classification —that is, the number of test images for which the system's top five predictions did not include the correct answer—fell from over 26% to less than 2.3% in just six years. The images are fairly large (e.g., 256 pixels high) and there are 1,000 classes, some of which are really subtle (try distinguishing 120 dog breeds).

Looking at the evolution of the winning entries is a good way to understand how CNNs work, and how research in deep learning progresses. We will first look at:

- LeNet-5 architecture (1998)

- AlexNet (2012),
- GoogLeNet (2014),
- ResNet (2015),
- SENet (2017).

In addition, we will also briefly look at more architectures, including Xception, ResNeXt, DenseNet, MobileNet, CSPNet, and EfficientNet.

2.6.1 LeNet-5

The LeNet-5 architecture is perhaps the most widely known CNN architecture. As mentioned, it was created by Yann LeCun in 1998 and has been widely used for handwritten digit recognition (MNIST).

Layer	Type	Maps	Size	Kernel Size
Out	Fully Connected	-	10	-
F6	Fully connected	-	84	-
C5	Convolution	120	1-by-1	5-by-5
S4	Average Pooling	16	5-by-5	2-by-2
C3	Convolution	16	10-by-10	5-by-5
S2	Average Pooling	6	14-by-14	2-by-2
C1	Convolution	6	28-by-28	5-by-5
In	Input	1	32-by-32	-

Table 2.1.: LeNet-5 Architecture.

As we can see, this looks pretty similar to our Fashion MNIST model: a stack of convolutional layers and pooling layers, followed by a dense network. Perhaps the main difference with more modern classification CNNs is the activation functions: today, we would use ReLU instead of tanh and softmax instead of RBF.

2.6.2 AlexNet

The AlexNet CNN architecture won the 2012 ILSVRC challenge by a large margin: it achieved a top-five error rate of 17%, while the second best competitor achieved only 26%! AlexaNet was developed by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. It is similar to LeNet-5, only much larger and deeper, and it was the first to stack convolutional layers directly on top of one another, instead of stacking a pooling layer on top of each convolutional layer. Table X presents this architecture.

To reduce overfitting, the authors used two regularization techniques. First, they applied dropout with a 50% dropout rate during training to the outputs of layers F9 and F10. Second, they performed data augmentation by randomly shifting the training images by various offsets, flipping them horizontally, and changing the lighting conditions.

Data Augmentation

It is the process of artificially increasing the size of the training set by generating many realistic **variants** of each training instance. This process aims to reduce over-fitting, making this a regularisation technique.

The generated instances should be as realistic as possible. This means, in an ideal scenario, a human should not be able to tell whether it was augmented or not. In adition, it has to be something learnable; Adding white noise will not help as it is a random process and there is no pattern in the data for the algorithm to learn.

For example, we can slightly shift, rotate, and resize every picture in the training set by various amounts and add the resulting pictures to the training set.

Figure 2.13.: Data augmentation is the process of artificially generating new data from existing data. Here can see the process where the original image is transformed (shear, rotation) and fed to the ML to train on this new data. This allows the reuse of the image without requiring to gather new data [2].

To use this feature in code, use Keras's data augmentation layers, (i.e., `RandomCrop`, `RandomRotation`, etc.). These layers force the model to be more tolerant to variations in the position, orientation, and size of the objects in the pictures. To produce a model that's more tolerant of different lighting conditions, we can similarly generate many images with various contrasts. In general, we can also flip the pictures horizontally (except for text, and other asymmetrical objects). By combining these transformations, we can greatly increase your training set size. Example of this operation can be seen in **Fig.** 2.13 .

Data augmentation is also useful when we have an unbalanced dataset: we can use it to generate more samples of the less frequent classes. This is called the synthetic minority oversampling technique (SMOTE).

AlexNet also uses a competitive normalization step immediately after the ReLU step of layers C1 and C3, called *local response normalization* (LRN):

> The most strongly activated neurons inhibit other neurons located at the same position in neighboring feature maps.

Such competitive activation has been observed in biological neurons [7]. This encourages different feature maps to specialize, pushing them apart and forcing them to explore a wider range of features, ultimately improving generalization. The following equation gives use a view on how to apply this method:

$$b_i = a_i \left(k + \alpha \sum_{j=j_{low}}^{j_{high}} a_j^2 \right)^{-\beta} \qquad \text{with} \qquad \begin{cases} j_{high} = \min\left(i + \frac{r}{2},\ f_n - 1\right) \\ j_{low} = \max\left(0,\ i - \frac{r}{2}\right) \end{cases}$$

Let's discuss what each parameter means:

- b_i is the neuron's normalised output located in feature map i, at some row u and column v [17].
- a_i is the activation of that neuron after the ReLU step, but before normalisation.
- k, α, β and r are hyperparameters. k is called the bias, and r is called the depth radius.
- f_n is the number of feature maps.

[17] In this equation we consider only neurons located at this row and column, so u and v are not shown

To give an example, if $r = 2$ and a neuron has a **strong activation**, it will inhibit the activation of the neurons located in the feature maps immediately above and below its own.

In AlexNet, the hyperparameters are set as:

$$r = 5,\ \alpha = 0.0001,\ \beta = 0.75 \quad \text{and} \quad k = 2.$$

We can implement this step by using the `tf.nn.local_response_normalization()` function.

> A variant of AlexNet called ZF Net was developed by Matthew Zeiler and Rob Fergus and won the 2013 ILSVRC challenge [44]. It is essentially AlexNet with a few tweaked hyperparameters (number of feature maps, kernel size, stride, etc.).

2.6.3 GoogLeNet

The GoogLeNet architecture was developed by Christian Szegedy et al. from Google Research, and won the ILSVRC 2014 challenge by pushing the top-five error rate below 7% [38].

This performance boost came in from the network being deeper than previous CNNs. This was made possible by subnetworks called inception modules, which allow GoogLeNet to use parameters much more efficiently than previous architectures:

> GoogLeNet actually has 10 times fewer parameters than AlexNet[18].

[18] roughly 6 million instead of 60 million

Figure 14-14 shows the architecture of an inception module. The notation "$3 \times 3 + 1\,(S)$" means the layer uses a 3-by-3 kernel, stride 1, and `same` padding. The input signal is first fed to four (4) different layers in parallel. All convolutional layers use the ReLU activation function.

> Top convolutional layers use different kernel sizes (1-by-1, 3-by-3, and 5-by-5), allowing them to capture patterns at different scales.

Also note that every single layer uses a stride of 1 and "same" padding[19]. This is done so outputs all have the same height and width as their inputs. This allows concatenating all outputs along the depth dimension in the final depth concatenation layer (i.e., to stack the feature maps from all four top convolutional layers).

[19] even the max pooling layer

> It can be implemented using Keras's `Concatenate` layer, using the default `axis=-1`.

You may wonder why inception modules have convolutional layers with 1-by-1 kernels.

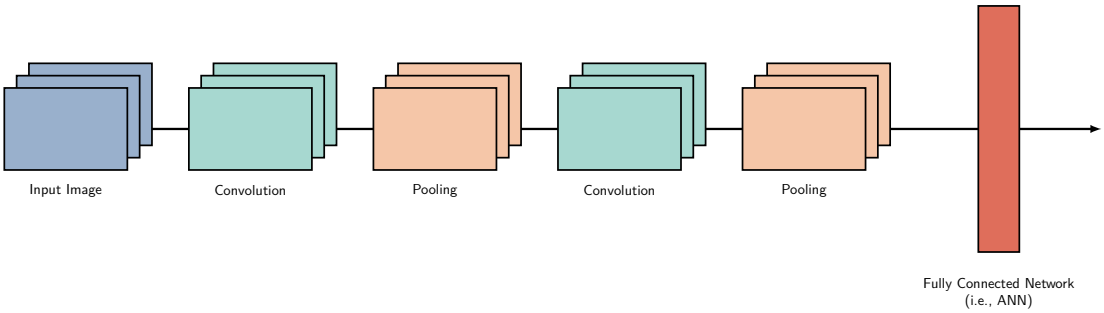

Figure 2.14.: The GoogLeNet architecture. The nodes coloured in (■) are called inception nodes.

Surely these layers cannot capture any features because they look at only one pixel at a time, right?

In fact, these layers serve three (3) purposes:

1. Although they cannot capture spatial patterns, they can capture patterns along the depth dimension (i.e., across channels).

2. They are configured to output fewer feature maps compared to their inputs, so they serve as **bottleneck layers**, meaning they reduce dimensionality. This cuts the computational cost and the number of parameters, speeding up training and improving generalization.

3. Each pair of convolutional layers ([1-by-1, 3-by-3] and [1-by-1, 5-by-5]) acts like a single powerful convolutional layer, capable of capturing more complex patterns. A convolutional layer is equivalent to sweeping a dense layer across the image (at each location, it only looks at a small receptive field), and these pairs of convolutional layers are equivalent to sweeping two-layer neural networks across the image.

In short, we can think of the whole inception module as a convolutional layer on steroids, able to output feature maps that capture complex patterns at various scales.
Now let's look at the architecture of the GoogLeNet CNN shown in **Fig.** 2.14 . The number of feature maps output by each convolutional layer and each pooling layer is shown before the kernel size. The architecture is so deep that it has to be represented in three (3) columns, but GoogLeNet is actually one tall stack, including nine (9) inception modules (nodes coloured in ■). The six numbers in the inception modules represent the number of feature maps output by each convolutional layer in the module.

> All the convolutional layers use the ReLU activation function.

Let's go through this network together:

- The first two (2) layers divide the image's height and width by 4 (so its area is divided by 16). This reduces the computational load. The first layer uses a large kernel size, 7-by-7, so that much of the information is preserved.

- Then the local response normalization layer ensures that the previous layers learn a wide variety of features .

Local Normalisation Layer

This layer's job is to create a sort of lateral inhibition. This refers to the capacity of an excited neuron to subdue its neighbors. We basically want a significant peak so that we have a form of local maxima. This tends to create a contrast in that area, hence increasing the sensory perception.

- Two convolutional layers follow, where the first acts like a bottleneck layer. Think of this pair as a single smarter convolutional layer.
- A local response normalisation layer ensures the previous layers capture a wide variety of patterns.
- Next, a max pooling layer reduces the image height and width by 2, again to speed up computations.
- Then comes the CNN's backbone: a tall stack of nine (9) inception modules, interleaved with a couple of max pooling layers to reduce dimensionality and speed up the net.
- Next, the global average pooling layer outputs the mean of each feature map: this drops any remaining spatial information, which is fine because there is not much spatial information left at that point. Indeed, GoogLeNet input images are typically expected to be 224 × 224 pixels, so after 5 max pooling layers, each dividing the height and width by 2, the feature maps are down to 7 × 7.

This classification task, not localization, so it doesn't matter where the object is.

Thanks to the dimensionality reduction brought by the global average pool layer, there is no need to have several fully connected layers at the top of the CNN[20], and this considerably reduces the number of parameters in the network and limits the risk of overfitting.

[20] This is unlike AlexNet.

- The last layers are self-explanatory: dropout for regularization, then a fully connected layer with 1,000 units (since there are 1,000 classes) and a softmax activation function to output estimated class probabilities.

The original GoogLeNet architecture included two auxiliary classifiers plugged on top of the third and sixth inception modules. They were both composed:

- One average pooling layer
- one convolutional layer
- two fully connected layers
- a softmax activation layer

During training, their loss (scaled down by 70%) was added to the overall loss.

The goal adding these auxiliary classifiers was to fight the vanishing gradients problem and regularize the network, but it was later shown that their effect was relatively minor.

Several variants of the GoogLeNet architecture were later proposed by Google researchers, including Inception-v3 [41] and Inception-v4 [39], using slightly different inception modules to reach even better performance.

2.6.4 VGGNet

The runner-up in the ILSVRC 2014 challenge was VGGNet [22], Karen Simonyan and Andrew Zisserman, from the Visual Geometry Group (VGG) research lab at Oxford University, developed a very simple and classical architecture; it had 2 or 3 convolutional layers and a pooling layer, then again 2 or 3 convolutional layers and a pooling layer, and so on, reaching a total of 16 or 19 convolutional layers, depending on the VGG variant. To add to this stack a final dense network with 2 hidden layers and the output layer. It used small 3-by-3 filters, but it had many of them.

2.6.5 ResNet

Kaiming He et al. won the ILSVRC 2015 challenge using a Residual Network (ResNet) that delivered an astounding top-five error rate under 3.6%. The winning variant used an extremely deep CNN composed of 152 layers (other variants had 34, 50, and 101 layers) [13].

> Computer vision models are getting deeper and deeper, with fewer and fewer parameters.

The key idea for training such a deep network is to use skip connections[21].

> The signal feeding into a layer is also added to the output of a layer located higher up the stack.

[21] This is also called shortcut connections.

Let's see why this is useful. When training a neural network, the goal is simple:

> To make it model a target function $h(x)$.

If we add the input x to the output of the network (i.e., we add a skip connection), then the network will be forced to model:

$$f(x) = h(x) - x \qquad \text{rather than} \qquad h(x)$$

This approach is called residual learning [13].

When we initialise a regular neural network, its weights are close to zero[22], so the network just outputs values close to zero. If we add a skip connection, the resulting network just outputs a copy of its inputs. In other words, it initially models the identity function.

[22] They are not exactly zero but randomly assigned and are close to zero

> If the target function is fairly close to the identity function (which is often the case), this will speed up training considerably.

In addition to the previously mentioned positive aspects, if we add many skip connections, the network can start making progress even if several layers have not started learning yet [28], which we can see the diagram in **Fig.** 2.15 .

> Skip connections allows the signal to easily make its way across the whole network.

The deep residual network can be seen as a stack of residual units (RUs), where each residual unit is a small neural network with a skip connection. Now let's look at ResNet's architecture (see Figure 14-18). The idea of ResNet is simple to describe. It starts and ends exactly like GoogLeNet (except without a dropout layer), and in between is just a very deep stack of residual units. Each residual unit is composed of two (2) convolutional layers[23], with batch normalization (BN) and ReLU activation, using 3-by-3 kernels and preserving spatial dimensions (stride of 1, `"same"` padding).

[23] There are no pooling layers

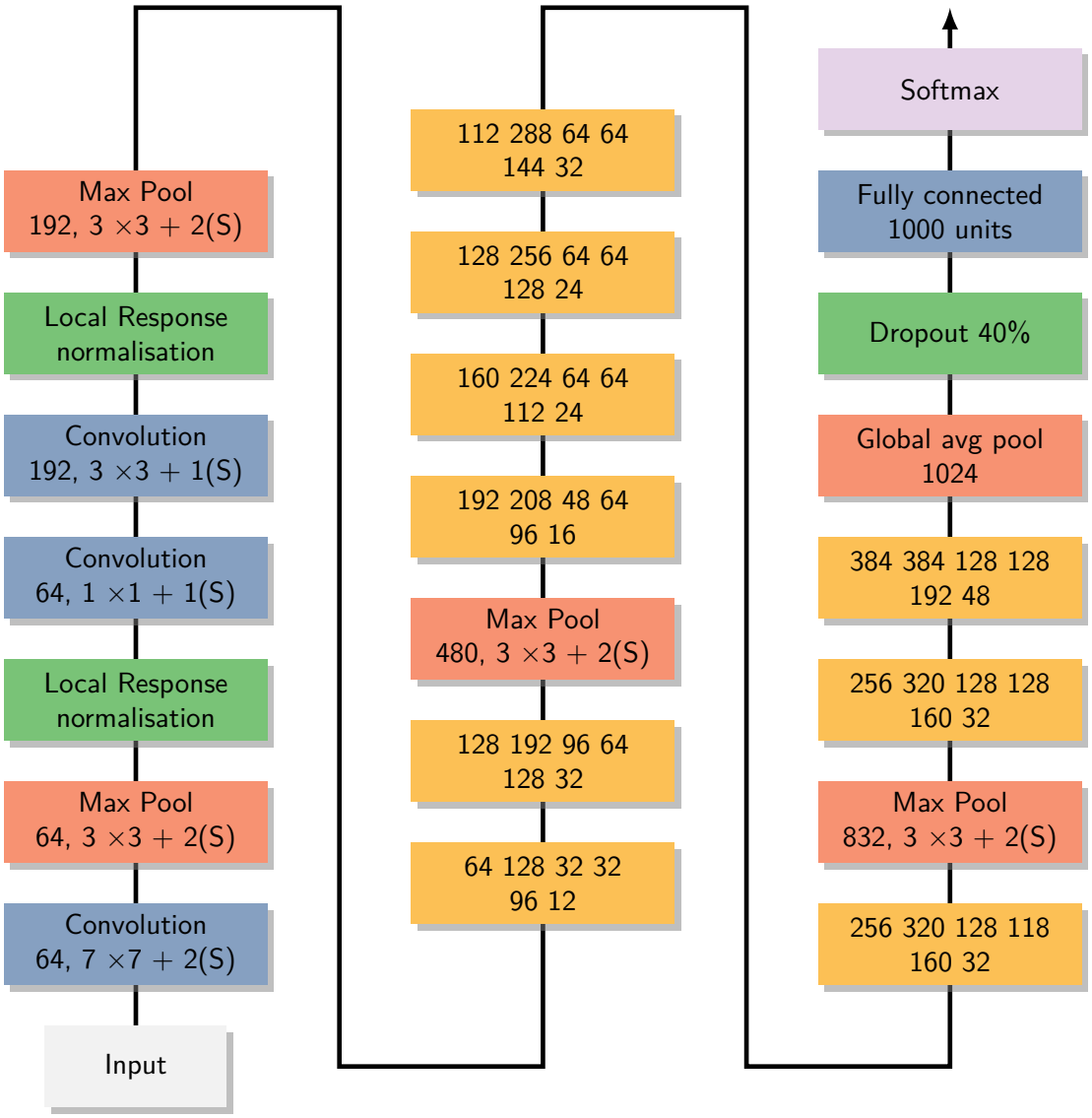

Figure 2.15.

> The number of feature maps is doubled every few residual units, at the same time as their height and width are halved (using a convolutional layer with stride 2)

When this happens, the inputs cannot be added directly to the outputs of the residual unit because they don't have the same shape (for example, this problem affects the skip connection represented by the dashed arrow in Figure 14-18). To solve this problem, the inputs are passed through a 1-by-1 convolutional layer with stride 2 and the right number of output feature maps (see Figure 14-19).

There are different variations of this aforementioned architecture, each having different numbers of layers. ResNet-34, as the name implies, is a ResNet with 34 layers (only counting the convolutional layers and the fully connected layer) containing 3 RUs that output 64 feature maps, 4 RUs with 128 maps, 6 RUs with 256 maps, and 3 RUs with 512 maps.

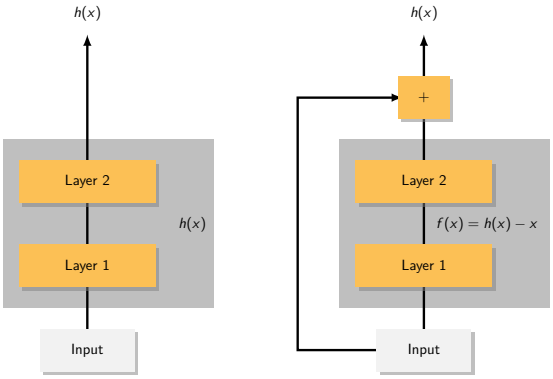

Figure 2.16.

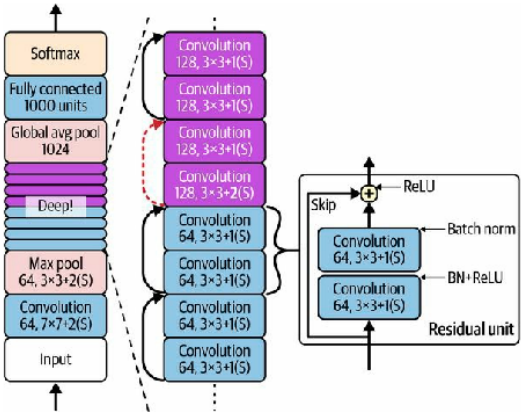

Figure 2.17.

ResNets deeper than that, such as ResNet-152, use slightly different residual units. Instead of two (2) 3-by-3 convolutional layers with 256 feature maps, they use three (3) convolutional layers:

1. a 1-by-1 convolutional layer with just 64 feature maps (4x less), which acts as a bottleneck layer (as discussed already)
2. a 3-by-3 layer with 64 feature maps
3. another 1-by-1 convolutional layer with 256 feature maps (4 times 64) that restores the original depth

ResNet-152 contains 3 such RUs that output 256 maps, then 8 RUs with 512 maps, a whopping 36 RUs with 1,024 maps, and finally 3 RUs with 2,048 maps.

2.7 Implementing a ResNet-34 CNN using Keras

[24] although generally we would load a pre-trained network instead, as we will see later.

Most CNN architectures described so far can be implemented easily using Keras[24]. To illustrate the process, let's implement a ResNet-34 from scratch with Keras.

First, we'll create a `ResidualUnit` layer:

C.R. 11 python

```
DefaultConv2D = partial(tf.keras.layers.Conv2D, kernel_size=3, strides=1,
                        padding="same", kernel_initializer="he_normal",
                        use_bias=False)

class ResidualUnit(tf.keras.layers.Layer):
    def __init__(self, filters, strides=1, activation="relu", **kwargs):
        super().__init__(**kwargs)
        self.activation = tf.keras.activations.get(activation)
        self.main_layers = [
            DefaultConv2D(filters, strides=strides),
            tf.keras.layers.BatchNormalization(),
            self.activation,
            DefaultConv2D(filters),
            tf.keras.layers.BatchNormalization()
        ]
        self.skip_layers = []
        if strides > 1:
            self.skip_layers = [
                DefaultConv2D(filters, kernel_size=1, strides=strides),
                tf.keras.layers.BatchNormalization()
            ]

    def call(self, inputs):
        Z = inputs
        for layer in self.main_layers:
            Z = layer(Z)
        skip_Z = inputs
        for layer in self.skip_layers:
            skip_Z = layer(skip_Z)
        return self.activation(Z + skip_Z)

```

As we can see, this code matches Figure 14-19 pretty closely. In the constructor, we create all the layers we will need:

> the main layers are the ones on the right side of the diagram, and the skip layers are the ones on the left[25].

[25] only needed if the stride is greater than 1.

Then in the `call()` method, we make the inputs go through the main layers and the skip layers (*if any*), and we add both outputs and apply the activation function.

Now we can build a ResNet-34 using a Sequential model, as it's really just a long sequence of layers. We can treat each residual unit as a single layer now that we have the `ResidualUnit` class.

The code closely matches:

C.R. 12 python

```
model = tf.keras.Sequential([
    DefaultConv2D(64, kernel_size=7, strides=2, input_shape=[224, 224, 3]),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Activation("relu"),
    tf.keras.layers.MaxPool2D(pool_size=3, strides=2, padding="same"),
])
prev_filters = 64
```

C.R. 13 python
```
for filters in [64] * 3 + [128] * 4 + [256] * 6 + [512] * 3:
    strides = 1 if filters == prev_filters else 2
    model.add(ResidualUnit(filters, strides=strides))
    prev_filters = filters

model.add(tf.keras.layers.GlobalAvgPool2D())
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(10, activation="softmax"))
```

The only tricky part in this code is the loop that adds the `ResidualUnit` layers to the model. As explained earlier:

- the first 3 RUs have 64 filters,
- the next 4 RUs have 128 filters.

and so on. At each iteration, we must set the stride to 1 when the number of filters is the same as in the previous RU, or else we set it to 2; then we add the ResidualUnit, and finally we update `prev_filters`.

With just 40 lines of code, we can build the model that won the ILSVRC 2015 challenge. This demonstrates both the elegance of the ResNet model and the expressiveness of the Keras API. Implementing the other CNN architectures is a bit longer, but not much harder.

However, Keras comes with several of these architectures built in, so why not use them instead?

2.8 Using Pre-Trained Models from Keras

In general, we don't have to implement standard models like GoogLeNet or ResNet manually, as pre-trained networks are readily available using the `tf.keras.applications` package.

For example, we can load the ResNet-50 model, pre-trained on ImageNet, with the following line of code:

C.R. 14 python
```
model = tf.keras.applications.ResNet50(weights="imagenet")
```

This was surprisingly simple. This will create a ResNet-50 model and download weights already trained on the ImageNet dataset. To use it, we first need to ensure the images have the correct size. A ResNet-50 model expects an image with the dimensions of 224-by-224–pixel[26], so let's use the Resizing layer, provided by Keras, to resize two (2) sample images (after cropping them to the target aspect ratio):

[26] other models may expect other sizes, such as 299-by-299.

C.R. 15 python
```
K = tf.keras.backend
images = K.constant(load_sample_images()["images"])
images_resized = tf.keras.layers.Resizing(height=224, width=224,
                                          crop_to_aspect_ratio=True)(images)
```

The pre-trained models assumes the images are pre-processed in a specific way. In some cases the models can expect the inputs to be scaled from 0 to 1, or from –1 to 1, and so on. Each model provides

a `preprocess_input()` function we can use to pre-process our images. These functions assume the original pixel values range from 0 to 255, which is the case here:

C.R. 16
```python
inputs = tf.keras.applications.resnet50.preprocess_input(images_resized)
```

Now we can use the pre-trained model to make predictions:

C.R. 17
```python
Y_proba = model.predict(inputs)
print(Y_proba.shape)
```

```text
(2, 1000)
```

As usual, the output `Y_proba` is a matrix with one row per image and one column per class[27]. To display the top K predictions, including the class name and the estimated probability of each predicted class, use the `decode_predictions()` function. For each image, it returns an array containing the top `K` predictions, where each prediction is represented as an array containing the class identifier, its name, and the corresponding confidence score:

[27] in this case, there are 1,000 classes

C.R. 18
```python
top_K = tf.keras.applications.resnet50.decode_predictions(Y_proba, top=3)
for image_index in range(len(images)):
    print(f"Image #{image_index}")
    for class_id, name, y_proba in top_K[image_index]:
        print(f"  {class_id} - {name:12s} {y_proba:.2%}")
```

The output looks like this:

```text
Downloading data from
↪ https://storage.googleapis.com/download.tensorflow.org/data/imagenet_class_index.json
Image #0
  n03877845 - palace       54.69%
  n03781244 - monastery    24.71%
  n02825657 - bell_cote    18.55%
Image #1
  n04522168 - vase         32.67%
  n11939491 - daisy        17.82%
  n03530642 - honeycomb    12.04%
```

The correct classes are **palace** and **dahlia** (which you can see in **Fig.** 2.18), so the model is **correct for the first image but wrong for the second**.

> This is caused by dahlia not being part of the 1,000 ImageNet classes.

Keeping this in mind, vase is a reasonable guess[28], and daisy is also not a bad choice either, are dahlias and daisies both share similar features. As we can see, it is very easy to create a pretty good image classifier using a pre-trained model.

[28] The intricate design of the flower might be reinterpreted as a decoration painted on a vase

Many vision models are available in `tf.keras.applications`, from lightweight and fast models to large and accurate ones. But what if we want to use an image classifier for classes of images that are

Figure 2.18.: The images used in testing the image recognition.

not part of ImageNet? In that case, we may still benefit from the pre-trained models by using them to perform **transfer learning**.

2.9 Pre-Trained Models for Transfer Learning

If we want to build an image classifier but not have enough data to train it from scratch, it is often a good idea to reuse the lower layers of a pre-trained model [12]. For example, let's train a model to classify pictures of flowers, reusing a pre-trained Xception model.

First, we'll load the flowers dataset using TensorFlow Datasets:

C.R. 19 python
```
import tensorflow_datasets as tfds

dataset, info = tfds.load("tf_flowers", as_supervised=True, with_info=True)
dataset_size = info.splits["train"].num_examples
class_names = info.features["label"].names
n_classes = info.features["label"].num_classescp.store_fig("flower-set", close=True)
```

> We can get information about the dataset by setting `with_info=True`.

Here, we get the dataset size and the names of the classes. Unfortunately, there is only a `"train"` dataset, no test set or validation set, so we need to split the training set. Let's call `tfds.load()` again, but this time taking the first 10% of the dataset for testing, the next 15% for validation, and the remaining 75% for training:

C.R. 20 python
```
test_set_raw, valid_set_raw, train_set_raw = tfds.load(
    "tf_flowers",
    split=["train[:10%]", "train[10%:25%]", "train[25%:]"],
    as_supervised=True)
```

Figure 2.19.: Sample images present in the dataset dataset. As you can see the images are not all in the same shape which we need to work on.

All three (3) datasets contain individual images. First let's have a look at the data to get some idea on what we are working with.

We need to batch them, but first we need to ensure they all have the same size, otherwise batching will fail. We can use a `Resizing` layer for this. We must also call the `tf.keras.applications.xception.preprocess_input()` function to preprocess the images appropriately for the Xception model. Lastly, we'll also shuffle the training set and use prefetching:

C.R. 21 python

```
batch_size = 32
preprocess = tf.keras.Sequential([
    tf.keras.layers.Resizing(height=224, width=224, crop_to_aspect_ratio=True),
    tf.keras.layers.Lambda(tf.keras.applications.xception.preprocess_input)
])
train_set = train_set_raw.map(lambda X, y: (preprocess(X), y))
train_set = train_set.shuffle(1000, seed=42).batch(batch_size).prefetch(1)
valid_set = valid_set_raw.map(lambda X, y: (preprocess(X), y)).batch(batch_size)
test_set = test_set_raw.map(lambda X, y: (preprocess(X), y)).batch(batch_size)
```

Now each batch contains 32 images, all of them 224-by-224 pixels, with pixel values ranging from –1 to 1.

This is the ideal values for training such a network. As the dataset is not very large, a bit of data augmentation will certainly help. Let's create a data augmentation model that we will embed in our final model. During training, it will randomly flip the images horizontally, rotate them a little bit, and tweak the contrast:

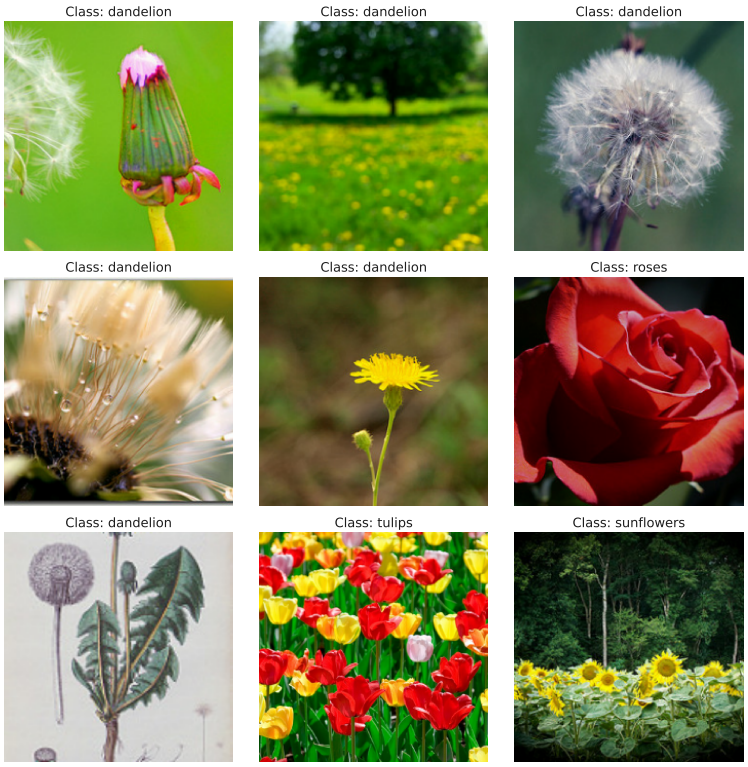

Figure 2.20.: Sample images present in the dataset, normalised and all of them have the same dimensions.

C.R. 22 python
```
data_augmentation = tf.keras.Sequential([
    tf.keras.layers.RandomFlip(mode="horizontal", seed=42),
    tf.keras.layers.RandomRotation(factor=0.05, seed=42),
    tf.keras.layers.RandomContrast(factor=0.2, seed=42)
])
```

Next let's load an Xception model, which is pre-trained on ImageNet. We exclude the top of the network by setting `include_top=False`. This excludes the global average pooling layer and the dense output layer. We then add our own global average pooling layer (feeding it the output of the base model), followed by a dense output layer with one unit per class, using the softmax activation function. Finally, we wrap all this in a Keras Model:

C.R. 23 python
```
tf.random.set_seed(42)  # extra code - ensures reproducibility
base_model = tf.keras.applications.xception.Xception(weights="imagenet",
                                                     include_top=False)
avg = tf.keras.layers.GlobalAveragePooling2D()(base_model.output)
output = tf.keras.layers.Dense(n_classes, activation="softmax")(avg)
model = tf.keras.Model(inputs=base_model.input, outputs=output)
```

It's usually a good idea to freeze the weights of the pre-trained layers, at least at the beginning of training[29].

[29] By not updating the weights of the frozen layers, we avoid tweaking features that are already well-established and generalize well across different tasks.

C.R. 24 python
```python
for layer in base_model.layers:
    layer.trainable = False
```

Finally, we can compile the model and start training:

C.R. 25 python
```python
optimizer = tf.keras.optimizers.SGD(learning_rate=0.1, momentum=0.9)
model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,
              metrics=["accuracy"])
history = model.fit(train_set, validation_data=valid_set, epochs=3)
```

After training the model for a few epochs, its validation accuracy should reach a bit over 80% and then stop improving. This means that the top layers are now pretty well trained, and we are ready to unfreeze some of the base model's top layers, then continue training.

For example, let's unfreeze layers 56 and above (that's the start of residual unit 7 out of 14, as you can see if you list the layer names):

C.R. 26 python
```python
for layer in base_model.layers[56:]:
    layer.trainable = True
```

Don't forget to compile the model whenever you freeze or unfreeze layers. Also make sure to use a much lower learning rate to avoid damaging the pre-trained weights:

C.R. 27 python
```python
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01, momentum=0.9)
model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,
              metrics=["accuracy"])
history = model.fit(train_set, validation_data=valid_set, epochs=10)
```

This model should reach around 92% accuracy on the test set, in just a few minutes of training (with a GPU). If you tune the hyperparameters, lower the learning rate, and train for quite a bit longer, you should be able to reach 95% to 97%.

But there's more to computer vision than just classification. For example, what if you also want to know where the flower is in a picture? Let's look at this now.

Bibliography

[1] S Agatonovic-Kustrin and Rosemary Beresford. "Basic concepts of artificial neural network (ANN) modeling and its application in pharmaceutical research". In: *Journal of pharmaceutical and biomedical analysis* 22.5 (2000), pp. 717–727.

[2] V. Alto. *Data Augmentation in Deep Learning*. 2020. URL: https://medium.com/analytics-vidhya/data-augmentation-in-deep-learning-3d7a539f7a28.

[3] David GT Barrett, Ari S Morcos, and Jakob H Macke. "Analyzing biological and artificial neural networks: challenges with opportunities for synergy?" In: *Current opinion in neurobiology* 55 (2019), pp. 55–64.

[4] BBC. *How a kingfisher helped reshape Japan's bullet train*. 2019. URL: https://www.bbc.com/news/av/science-environment-47673287.

[5] George Bebis and Michael Georgiopoulos. "Feed-forward neural networks". In: *Ieee Potentials* 13.4 (1994), pp. 27–31.

[6] Hans-Dieter Block. "The perceptron: A model for brain functioning. i". In: *Reviews of Modern Physics* 34.1 (1962), p. 123.

[7] Gustavo Deco and Edmund T Rolls. "Neurodynamics of biased competition and cooperation for attention: a model with spiking neurons". In: *Journal of neurophysiology* 94.1 (2005), pp. 295–313.

[8] Ronald A Fisher. "The use of multiple measurements in taxonomic problems". In: *Annals of eugenics* 7.2 (1936), pp. 179–188.

[9] Kunihiko Fukushima. "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position". In: *Biological cybernetics* 36.4 (1980), pp. 193–202.

[10] Wulfram Gerstner and Werner M Kistler. "Mathematical formulations of Hebbian learning". In: *Biological cybernetics* 87.5 (2002), pp. 404–415.

[11] Ian J Goodfellow, Oriol Vinyals, and Andrew M Saxe. "Qualitatively characterizing neural network optimization problems". In: *arXiv preprint arXiv:1412.6544* (2014).

[12] Xu Han et al. "Pre-trained models: Past, present and future". In: *AI Open* 2 (2021), pp. 225–250.

[13] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[14] Donald Olding Hebb. *The organization of behavior: A neuropsychological theory*. Psychology press, 2005.

[15] Robert Hecht-Nielsen. "Theory of the backpropagation neural network". In: *Neural networks for perception*. Elsevier, 1992, pp. 65–93.

[16] Dan Hendrycks and Kevin Gimpel. "Gaussian error linear units (gelus)". In: *arXiv preprint arXiv:1606.08415* (2016).

[17] Geoffrey Hinton et al. “Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups”. In: *IEEE Signal processing magazine* 29.6 (2012), pp. 82–97.

[18] Sean D Holcomb et al. “Overview on deepmind and its alphago zero ai”. In: *Proceedings of the 2018 international conference on big data and education*. 2018, pp. 67–71.

[19] Jim Howe. “Artificial intelligence at edinburgh university: A perspective”. In: *Archived from the original on* 17 (2007).

[20] Zhao Huang and Wei Zhao. “Combination of ELMo representation and CNN approaches to enhance service discovery”. In: *IEEE Access* 8 (2020), pp. 130782–130796.

[21] David H Hubel and Torsten N Wiesel. “Receptive fields, binocular interaction and functional architecture in the cat’s visual cortex”. In: *The Journal of physiology* 160.1 (1962), p. 106.

[22] Andrej Karpathy et al. “Large-scale video classification with convolutional neural networks”. In: *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*. 2014, pp. 1725–1732.

[23] Hinkelmann Knut. “Neural Networks p. 7”. In: *University of Applied Sciences Northwestern Switzerland* (2018).

[24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems* 25 (2012).

[25] Yann LeCun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[26] Qinglong Li et al. “A hybrid CNN-based review helpfulness filtering model for improving e-commerce recommendation Service”. In: *Applied Sciences* 11.18 (2021), p. 8613.

[27] Seppo Linnainmaa. “Algoritmin kumulatiivinen pyöristysvirhe yksittäisten pyöristysvirheiden Taylor-kehitelmänä”. Available in Finnish at `https://people.idsia.ch/~juergen/linnainmaa1970thesis.pdf`. Master’s thesis. University of Helsinki, 1970.

[28] Fenglin Liu et al. “Rethinking skip connection with layer normalization in transformers and resnets”. In: *arXiv preprint arXiv:2105.07205* (2021).

[29] Warren S McCulloch and Walter Pitts. “A logical calculus of the ideas immanent in nervous activity”. In: *The bulletin of mathematical biophysics* 5 (1943), pp. 115–133.

[30] Brilian Tafjira Nugraha, Shun-Feng Su, et al. “Towards self-driving car using convolutional neural network and road lane detector”. In: *2017 2nd international conference on automation, cognitive science, optics, micro electro-mechanical system, and information technology (ICACOMIT)*. IEEE. 2017, pp. 65–69.

[31] Zhenchao Ouyang et al. “Deep CNN-based real-time traffic light detector for self-driving vehicles”. In: *IEEE transactions on Mobile Computing* 19.2 (2019), pp. 300–313.

[32] Marius-Constantin Popescu et al. “Multilayer perceptron and neural networks”. In: *WSEAS Transactions on Circuits and Systems* 8.7 (2009), pp. 579–588.

[33] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. “Learning representations by back-propagating errors”. In: *nature* 323.6088 (1986), pp. 533–536.

[34] Wojciech Samek et al. “Explaining deep neural networks and beyond: A review of methods and applications”. In: *Proceedings of the IEEE* 109.3 (2021), pp. 247–278.

[35] Arnau Sebé-Pedrós. "Stepwise emergence of the neuronal gene expression program in early animal evolution". In: (2023).

[36] Sagar Sharma, Simone Sharma, and Anidhya Athaiya. "Activation functions in neural networks". In: *Towards Data Sci* 6.12 (2017), pp. 310–316.

[37] Haim Sompolinsky. "The theory of neural networks: The Hebb rule and beyond". In: *Heidelberg Colloquium on Glassy Dynamics: Proceedings of a Colloquium on Spin Glasses, Optimization and Neural Networks Held at the University of Heidelberg June 9–13, 1986*. Springer. 2006, pp. 485–527.

[38] Christian Szegedy et al. "Going deeper with convolutions". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 1–9.

[39] Christian Szegedy et al. "Inception-v4, inception-resnet and the impact of residual connections on learning". In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 31. 1. 2017.

[40] Julian FV Vincent et al. "Biomimetics: its practice and theory". In: *Journal of the Royal Society Interface* 3.9 (2006), pp. 471–482.

[41] Xiaoling Xia, Cui Xu, and Bing Nan. "Inception-v3 for flower classification". In: *2017 2nd international conference on image, vision and computing (ICIVC)*. IEEE. 2017, pp. 783–787.

[42] Hao Ye et al. "Evaluating two-stream CNN for video classification". In: *Proceedings of the 5th ACM on International Conference on Multimedia Retrieval*. 2015, pp. 435–442.

[43] Ákos Zarándy et al. "Overview of CNN research: 25 years history and the current trends". In: *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE. 2015, pp. 401–404.

[44] MD Zeiler. "Visualizing and Understanding Convolutional Networks". In: *European conference on computer vision/arXiv*. Vol. 1311. 2014.

[45] Wang Zhiqiang and Liu Jun. "A review of object detection based on convolutional neural network". In: *2017 36th Chinese control conference (CCC)*. IEEE. 2017, pp. 11104–11109.