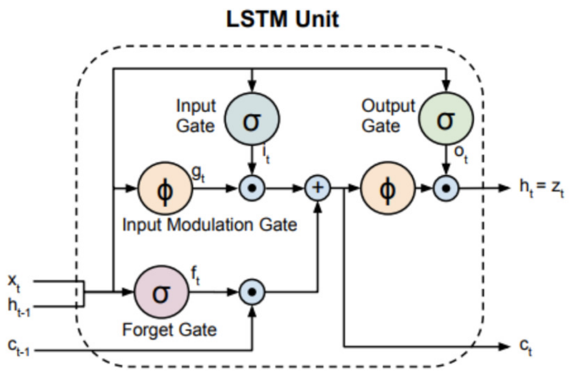

Figure 2.4: Illustration of a typical LSTM unit. The unit takes input at the current time, x_t, and from a previous time, h_{t-1}, and it returns an output to be fed into the next time, h_t. The final output of the LSTM unit is controlled by the input gate, i_t, the forget gate, f_t, and the output gate, o_t, as well as the memory cell state, c_t, which are defined in (2.5), (2.6), (2.9) and (2.8), respectively. Figure reproduced from [33].

2.1.3 Convolutional networks

Convolutional networks (ConvNets) are a special type of neural network that are especially well adapted to computer vision applications because of their ability to hierarchically abstract representations with local operations. There are two key design ideas driving the success of convolutional architectures in computer vision. First, ConvNets take advantage of the 2D structure of images and the fact that pixels within a neighborhood are usually highly correlated. Therefore, ConvNets eschew the use of one-to-one connections between all pixel units (*i.e.* as is the case of most neural networks) in favor of using grouped local connections. Further, ConvNet architectures rely on feature sharing and each channel (or output feature map) is thereby generated from convolution with the same filter at all locations as depicted in Figure 2.5. This important characteristic of ConvNets leads to an architecture that relies on far fewer parameters compared to standard Neural Networks. Second, ConvNets also introduce a pooling step that provides a degree of translation invariance making the architecture less affected by small variations in position. Notably, pooling also allows the network to gradually see larger portions of the input thanks to an increased size of the network's receptive field. The increase in receptive field size (coupled with a decrease in the input's resolution) allows the network to represent more abstract characteristics of the input as the network's depth increase. For example, for the task of object recognition, it is advocated that ConvNets layers start by focusing on edges to parts of the object to finally cover the entire object at higher layers in the hierarchy.

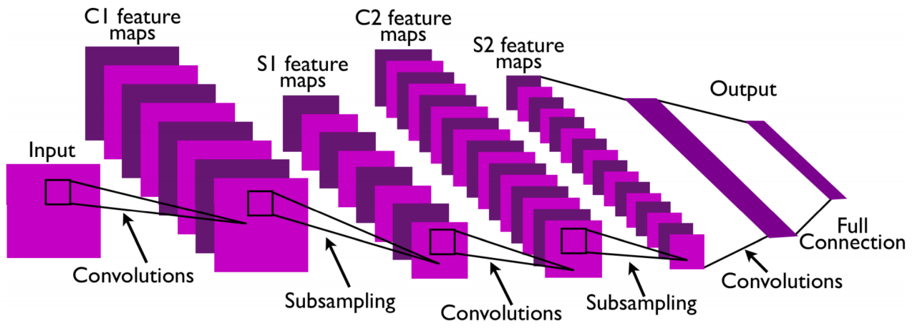

Figure 2.5: Illustration of the structure of a standard Convolutional Network. Figure reproduced from [93].

The architecture of convolutional networks is heavily inspired by the processing that takes place in the visual cortex as described in the seminal work of Hubel and Wiesel [74] (further discussed in Chapter 3). In fact, it appears that the earliest instantiation of Convolutional Networks is Fukushima's Neocognitron [49], which also relied on local connections and in which each feature map responds maximally to only a specific feature type. The Neocognitron is composed of a cascade of K layers where each layer alternates S-cell units, U_{sl}, and complex cell units, U_{cl}, that loosely mimic the processing that takes place in the biological simple and complex cells, respectively, as depicted in Figure 2.6. The simple cell units perform operations similar to local convolutions followed by a Rectified Linear Unit (ReLU) nonlinearity, $\varphi(x) = \begin{cases} x; & if\, x \geq 0 \\ 0; & x < 0 \end{cases}$,while the complex cells perform operations similar to average pooling. The model also included a divisive nonlinearity to accomplish something akin to normalization in contemporary ConvNets.

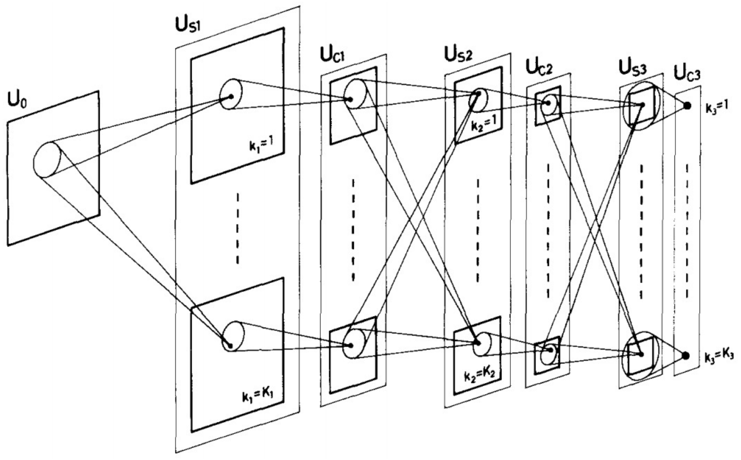

Figure 2.6: Illustration of the structure of the Neocognitron. Figure reproduced from [49].

As opposed to most standard ConvNet architectures (*e.g.* [88,91]) the Neocognitron does not need labeled data for learning as it is designed based on self organizing maps that learn the local connections between consecutive layers via repetitive presentations of a set of stimulus images. In particular, the Neocognitron is trained to learn the connections between an input feature map and a simple cell layer (connections between a simple cells layer and complex cells layer are pre-fixed) and the learning procedure can be broadly summarized in two steps. First, each time a new stimulus is presented at the the input, the simple cells that respond to it maximally are chosen as a representative cell for that stimulus type. Second, the connections between the input and those representative cells are reinforced each time they respond to the same input type. Notably, simple cells layers are organized in different groups or planes such that each plane responds only to one stimulus type (*i.e.* similar to feature maps in a modern ConvNet architecture). Subsequent extensions to the Neocognitron included allowances for supervised learning [51] as well as top-down attentional mechanisms [50].

Most ConvNets architectures deployed in recent computer vision applications are inspired by the successful architecture proposed by LeCun in 1998, now known as LeNet, for handwriting recognition [91]. As described in key literature [77,93], a classical convolutional network is made of four basic layers of processing: (i) a convolution layer, (ii) a nonlinearity or rectification layer, (iii) a normalization layer and (iv) a pooling layer. As noted above, these components were largely present in the Neocognitron. A key addition in LeNet was the incorporation of back propagation for relatively efficient learning of the convolutional parameters.

Although, ConvNets allow for an optimized architecture that requires far fewer parameters compared to their fully connected neural network counterpart, their main shortcoming remains their heavy reliance on learning and labeled data. This data dependence is probably one of the main reasons why ConvNets were not widely used until 2012 when the availability of the large ImageNet dataset [126] and concomitant computational resources made it possible to revive interest in ConvNets [88]. The success of ConvNets on ImageNet led to a spurt of various ConvNet architectures and most contributions in this field are merely based on different variations of the basic building blocks of ConvNets, as will be discussed later in Section 2.2.

2.1.4 Generative adversarial networks

Generative Adversarial Networks (GANs) are relatively new models taking advantage of the strong representational power of multilayer architectures. GANs were first introduced in 2014 [57] and although they did not present a different architecture per se (*i.e.* in terms of novel network building blocks for example), they entail some peculiarities, which make them a slightly different class of multilayer architectures. A key challenge being responded to by GANs is the introduction of an unsupervised learning approach that requires no labeled data.

A typical GAN is made of two competing blocks or sub-networks, as shown in Figure 2.7; a generator network, $G(\mathbf{z}; \theta_g)$, and a discriminator network, $D(\mathbf{x}; \theta_d)$, where $\mathbf{z}$ is input random noise, $\mathbf{x}$ is real input data (*e.g.* an image) and θ_g and θ_d are the parameters of the two blocks, respectively. Each block can be made of any of the previously defined multilayer architectures. In the original paper both the generator and discriminator were multilayer fully connected networks. The discriminator, D, is trained to recognize the data coming from the generator and assigning the label *"fake"* with probability p_d while assigning the label *"real"* to true input data with probability $1 - p_d$. In complement, the generator network is optimized to generate fake representations capable of fooling the discriminator. The

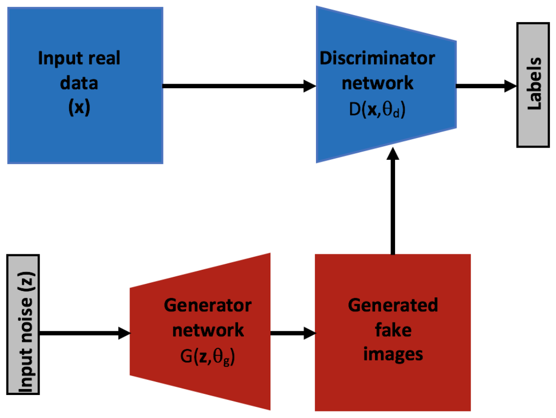

Figure 2.7: Illustration of the structure of a general purpose Generative Adverserial Network (GAN).

two blocks are trained alternately in several steps where the ideal outcome of the training process is a discriminator that assigns a probability of 50% to both real and fake data. In other words, after convergence the generator should be able to generate realistic data from random input.

Since the original paper, many contributions participated in enhancing the capabilities of GANs via use of more powerful multilayer architectures as the backbones of the network [114] (*e.g.* pretrained convolutional networks for the discriminator and deconvolutional networks, that learn upsampling filters for the generator). Some of the successful applications of GANs include: text to image synthesis (where the input to the network is a textual description of the image to be rendered [115]), image super resolution where the GAN generates a realistic high resolution image from a lower resolution input [94], image inpainting where the role of GANs is to fill holes of missing information from an input image [149] and texture synthesis where GANs are used to synthesize realistic textures from input noise [10].

2.1.5 Multilayer network training

As discussed in the previous sections, the success of the various multilayer architectures largely depends on the success of their learning process. While neural networks usually rely on an unsupervised pretraining step first, as described in Section 2.1.1, they are usually followed by the most widely used training strategy for multilayer architectures, which is fully supervised. The training procedure is usually based on error back propagation using gradient descent. Gradient descent is widely used in training multilayer architectures for its simplicity. It relies on minimizing a smooth error function, $E(\boldsymbol{w})$, following an iterative procedure defined as

$$\mathbf{w}_k = \mathbf{w}_{k-1} - \alpha \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}, \tag{2.11}$$

where $\mathbf{w}$ represents the network's parameters, α is the learning rate that may control the speed of convergence and $\frac{\partial \mathbf{E}(\mathbf{w})}{\partial \mathbf{w}}$ is the error gradient calculated over the training set. This simple gradient descent method is especially suitable for training multilayer networks thanks to the use of the chain rule for back propagating and calculating the error derivative with respect to various network's parameters at different layers. While back propagation dates back a number of years [16,146], it was popularized in the context of multilayer architectures [125]. In practice, stochastic gradient descent is used [2], which consists of approximating the error gradient over the entire training set from successive relatively small subsets.

One of the main problems of the gradient descent algorithm is the choice of the learning rate, α. A learning rate that is too small leads to slow convergence, while a large learning rate can lead to overshooting or fluctuation around the optimum. Therefore, several approaches were proposed to further improve the simple stochastic gradient descent optimization method. The simplest method, referred to as stochastic gradient descent with momentum [137], keeps track of the update amount from one iteration to another and gives momentum to the learning process by pushing the update further if the gradient keeps pointing to the same direction from one time step to another as defined in,

$$\mathbf{w}_k = \mathbf{w}_{k-1} - \alpha \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} - \gamma \left(\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}\right)_{t-1}, \tag{2.12}$$

with γ controlling the momentum. Another simple method involves setting the learning rate in a decreasing fashion according to a fixed schedule, but this is far from ideal given that this schedule has to be pre-set ahead of the training process and is completely independent from the data. Other more involved methods (*e.g.* Adagrad [34], Adadelta [152], Adam [86]) suggest adapting the learning rate during training to each parameter, w_i, being updated, by performing smaller updates on frequently changing parameters and larger updates on infrequent ones. A detailed comparison between the different versions of these algorithms can be found elsewhere [124].

The major shortcoming of training using gradient descent, as well as its variants, is the need for large amounts of labeled data. One way to deal with this difficulty is to resort to unsupervised learning. A popular unsupervised method used in training some shallow ConvNet architectures is based on the Predictive Sparse Decomposition (PSD) method [85]. Predictive Sparse Decomposition learns an overcomplete set of filters whose combination can be used to reconstruct an image. This method is especially suitable for learning the parameters of a convolutional architecture, as the algorithm is designed to learn basis functions that reconstruct an image patchwise. Specifically, Predictive Sparse Decomposition (PSD) builds on sparse coding algorithms that attempts to find an efficient representation, Y, of an input signal, X, via a linear combination with a basis set, B. Formally, the problem of sparse coding is broadly formulated as a minimization problem defined as,

$$L(X, Y; B) = ||X - BY||_2^2. \tag{2.13}$$

PSD adapts the idea of sparse coding in a convolutional framework by minimizing a reconstruction error defined as,

$$L(X, Y; B) = ||X - BY||_2^2 + \lambda ||Y||_1 + \alpha ||Y - F(X; G, W, D)||_2^2 \tag{2.14}$$

where $F(X; G, W, D) = G \tanh(WX + D)$ and W, D and G are weights, biases and gains (or normalization factors) of the network, respectively. By minimizing the loss function defined in equation 2.14, the algorithm learns a representation, Y, that reconstructs the input patch, X, while being similar to the predicted representation F. The learned representation will also be sparse owing to the second term of the equation. In practice, the error is minimized in two alternating steps where parameters, (B, G, W, D), are fixed and minimization is performed over Y. Then, the representation Y is fixed while minimizing over the other parameters. Notably, PSD is applied in a patchwise procedure where each set of parameters, (G, W, D), is learned from the reconstruction of a different patch from an input image. In other words, a different set of kernels is learned by focusing the reconstruction on different parts of the input images.

2.1.6 A word on transfer learning

One of the unexpected benefits of training multilayer architecture is the surprising adaptability of the learned features across different datasets and even different tasks. Examples include using networks trained with ImageNet for recognition on: other object recognition datasets such as Caltech-101 [38] (*e.g.* [96, 154]), other recognitions tasks such as texture recognition (*e.g.* [25]), other applications such as object detection (*e.g.* [53]) and even to video based tasks, such as video action recognition (*e.g.* [41, 134, 144]).

The adaptability of features extracted with multilayer architectures across different datasets and tasks, can be attributed to their hierarchical nature where the representations progress from being simple and local to abstract and global. Thus, features extracted at lower levels of the hierarchy tend to be common across different tasks thereby making multilayer architectures more amenable to transfer learning.

A systematic exploration of the intriguing transferability of features across different networks and tasks revealed several good practices to take into account in consideration of transfer learning [150]. First, it was shown that fine tuning higher layers only, led to systematically better performance when compared to fine tuning the entire network. Second, this research demonstrated that the more different the tasks are the less efficient transfer learning becomes. Third, and more surprisingly, it was found that even after fine tuning the network's performance under the initial task is not particularly hampered.

Recently, several emerging efforts attempt to enforce a networks' transfer learning capabilities even further by casting the learning problem as a sequential two step procedure, *e.g.* [3, 127]. First, a so called rapid learning step is performed where a network is optimized for a specific task as is usually done. Second, the network parameters are further updated in a global learning step that attempts to minimize an error across different tasks.

2.2 Spatial convolutional networks

In theory, convolutional networks can be applied to data of arbitrary dimensions. Their two dimensional instantiations are well suited to the structure of single images and therefore have received considerable attention in computer vision. With the availability of large scale datasets and powerful computers for training, the vision community has recently seen a surge in the use of ConvNets for various applications. This section describes the most prominent 2D ConvNet architectures that introduced relatively novel components to the original LeNet described in Section 2.1.3.

2.2.1 Key architectures in the recent evolution of ConvNets

The work that rekindled interest in ConvNet architectures was Krishevsky's AlexNet [88]. AlexNet was able to achieve record breaking object recognition results on the ImageNet dataset. It consisted of eight layers in total, 5 convolutional and 3 fully connected, as depicted in Figure 2.8.

AlexNet introduced several architectural design decisions that allowed for efficient training of the network using standard stochastic gradient descent. In particular, four important contributions were key to the success of AlexNet. First, AlexNet considered the use of the ReLU nonlinearity instead of the saturating nonlinearites, such as sigmoids, that were used in previous state-of-the-art ConvNet architectures (*e.g.* LeNet [91]). The use of the ReLU diminished the problem of vanishing gradient and led to faster training. Second, noting the fact that the last fully connected layers in a network contain the largest number of parameters, AlexNet used dropout, first introduced in the context of neural networks [136], to reduce the problem of overfitting. Dropout, as implemented in AlexNet, consists in randomly dropping (*i.e.* setting to zero) a given percentage of a layer's parameters. This technique allows for training a slightly different architecture at each pass and artificially reducing the number of parameters to be learned at each pass, which ultimately helps break correlations between units and thereby combats overfitting. Third, AlexNet relied on data augmentation to improve the network's ability to learn invariant representations. For example, the network was trained not only on the original images in the training set, but also on variations generated by randomly shifting and reflecting the training images. Finally, AlexNet also relied on several techniques to make the training process converge faster, such as the use momentum and a scheduled learning rate decrease whereby the learning rate is decreased every time the learning stagnates.

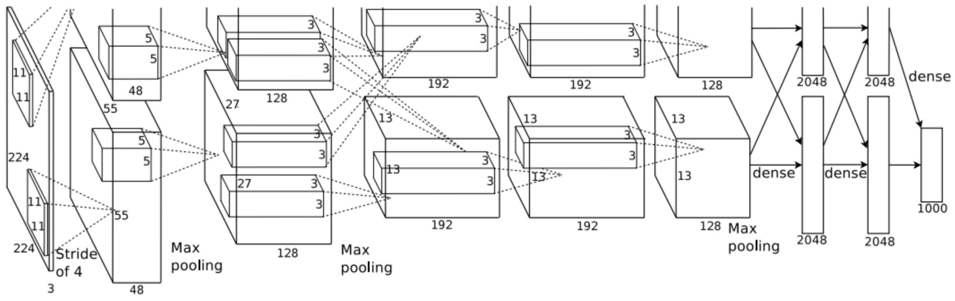

Figure 2.8: AlexNet architecture. Notably, although the depiction suggests a two stream architecture, it was in fact a single stream architecture and this depiction only reflects the fact that AlexNet was trained in parallel on 2 different GPUs. Figure reproduced from [88].

The advent of AlexNet led to a spurt in the number of papers trying to understand what the network is learning either via visualization, as done in the so called DeConvNet [154], or via systematic explorations of various architectures [22, 23]. One of the direct results of these explorations was the realization that deeper networks can achieve even better results as first demonstrated in the 19 layer deep VGG-Net [135]. VGG-Net achieves its depth by simply stacking more layers while following the standard practices introduced with AlexNet (*e.g.* reliance on the ReLU nonlinearity and data augmentation techniques for better training). The main novelty presented in VGG-Net was the use of filters with smaller spatial extent (*i.e.* 3×3 filters throughout the network instead of *e.g.* 11×11 filters used in AlexNet), which allowed for an increase in depth without dramatically increasing the number of parameters that the network needs to learn. Notably, while using smaller filters, VGG-Net required far more filters per layer.