

Neural Networks and Deep Learning Fundamentals



Riccardo Zese , Elena Bellodi , Michele Fraccaroli , Fabrizio Riguzzi ,
and Evelina Lamma 

Abstract In the last decade, Neural Networks (NNs) have come to the fore as one of the most powerful and versatile approaches to many machine learning tasks. Deep Learning (DL), the latest incarnation of NNs, is nowadays applied in every scenario that needs models able to predict or classify data. From computer vision to speech-to-text, DL techniques are able to achieve super-human performance in many cases. This chapter is devoted to give a (not comprehensive) introduction to the field, describing the main branches and model architectures, in order to try to give a roadmap of this area to the reader.

1 A Brief History

In the last decades, the use of Neural Networks (NNs) has become one of the most effective approaches for solving classification and regression tasks. This is principally due to their capability of identifying and modeling complex correlations. They have been proposed for the first time in the 40s in order to try to achieve two main objectives: to study the functioning of the human brain by defining models able to simulate its neuro-physiological phenomena, and to use such models to extract

R. Zese (✉)

Department of Chemical, Pharmaceutical and Agricultural Sciences, University of Ferrara,
Ferrara, Italy

e-mail: riccardo.zese@unife.it

E. Bellodi · M. Fraccaroli · E. Lamma

Department of Engineering, University of Ferrara, Ferrara, Italy

e-mail: elena.bellodi@unife.it

M. Fraccaroli

e-mail: michele.fraccaroli@unife.it

E. Lamma

e-mail: evelina.Lamma@unife.it

F. Riguzzi

Department of Mathematics and Computer Science, University of Ferrara, Ferrara, Italy

e-mail: fabrizio.riguzzi@unife.it

the principles guiding human reasoning in terms of mathematical calculations, to develop artificial systems able to reason as a human but possibly faster and more efficiently.

The concept of NN, as the name explicitly says, was originally used to define networks simulating the neurons and their interactions in the human brain. The first theory was developed by McCulloch and Walter Pitts [31] and was very simple and effective. They defined the neuron as a computational unit that applies a function to the inputs implementing a binary classification. The inputs were multiplied by weights fixed a priori and static. In 1958, this first definition was at the basis of the implementation of the first model of neural network, called Perceptron [39], allowing the training of a single neuron. From the work of those years, besides the Perceptron, a second learning algorithm also emerged, which was a special case of the *Stochastic Gradient Descent* (SGD) technique, which is at the basis of most learning algorithms now. This learning algorithm was used to train the weights of the ADaptive LINear Element (ADALINE) model [48], used for linear regression. Basically, SGD needs an error function that returns, for a given output of the model, how far the output is from the label of the input. SGD takes the error function and computes the gradient of this error on the weights. In this way, it is possible to update the weights moving along the gradient in order to reduce the error. Performing these operations iteratively allows minimizing the error. Nowadays, this idea has been declined in many ways, considering for example also the second derivative or the derivative of the previous iterations to better guide the tuning of the weights. However, a single neuron is not effective for even the simplest classification and regression task because it implements a too simple function. For this reason, this idea was abandoned at that time.

In the second half of the eighties, Rumelhart et al. [40] used the back-propagation mechanism to train larger networks, giving new life to the study of the topic. The idea was simple but effective: using more neurons means combining more functions, defining a model able to represent more complex scenarios. Unfortunately, at that time, developing and experimenting with these ideas was difficult due to hardware limitations that made the training unfeasible with the increase of the complexity of the model. However, the idea of *deep networks* dates back to these years. Deep networks are networks with many layers of neurons, the internal layers are called hidden and the depth of the network is the number of layers. Moreover, the eighties saw the first theorization of Convolutional Neural Networks (CNN) by Yan LeCun [27], whose work resulted in the definition of the well-known LeNet5 model [26]. This model was one of the first effective CNN as it was able to achieve super-human results in the task of handwritten digit recognition.

Nowadays, the idea of NNs is that of extremely complex networks with many neurons organized in many layers. The concept of depth is stressed even further, partly thanks to the advances in hardware. Other important aspects that helped to increase the importance of these models are the increase of the size of the data and the availability of powerful (and in many cases user-friendly) systems and frameworks

for the design, the implementation and the use of these models. Frameworks such as Tensorflow,¹ PyTorch² and Caffe³ allow a better user friendliness and consequently a easier prototyping of models.

The classical definition of a NN, a set of neurons grouped in different layers where a neuron in a layer communicates with all the neurons of the next layer, is now usually called Fully Connected (Deep) Neural Network, Artificial Neural Network, Multilayer Perceptron, or Deep Feedforward Network [1, 13], and has been later extended in order to define more complex models.

Alongside this definition, new models have gained increasing importance in the field. On one hand, the already mentioned convolutional networks, mainly used in the field of computer vision, and on the other hand Recurrent Neural Networks [40], defined for input data in the form of sequences, such as written text. Their great innovation is that they allow the network to maintain a memory of previous data, enabling the management of such sequences.

The rest of this chapter is organized as follow. Section 2 discusses Multilayer Perceptrons, Sect. 3 introduces Convolutional Neural Networks, and Sect. 4 Recurrent Neural Networks. Section 5 discusses the problem of tuning the hyper-parameters to guide the training. Finally, Sect. 6 concludes the chapter.

2 Multilayer Perceptrons

The objective of a Multilayer Perceptron (MLP) is that of approximating a function $\hat{f} : \mathbb{R}^{N \times M} \rightarrow \mathbb{R}^K$ by means of other functions such that $\hat{f}(X) = f_n(f_{n-1}(\dots(f_1(f_0(X))))))$, where X is a tensor⁴ of size $N \times M$ representing the input. The corresponding model is a network of n layers, each representing the function f_i with i the number of the layer. Figure 1 shows an example of MLP with 4 layers. Layer 0 is the input layer, the leftmost one, then layers 1 and 2 are internal layers, also called *hidden layers*. Finally, the last layer is the output one, returning the results of the computation of the network. The output function is $\hat{Y} = \hat{f}(X) = f_3(f_2(f_1(f_0(X))))$, with $\hat{f} : \mathbb{R}^{3 \times 1} \rightarrow \mathbb{R}^3$. This network is developed to solve a multiclass classification where each input x can be labelled with three different classes, associated with different neurons of the output layer. Before describing the overall flow, let us concentrate on a single neuron, e.g., neuron h_1 in Fig. 1. Figure 2 shows the computation performed in each single neuron. Basically, each neuron takes as input the output of each neurons of the previous layer, for the case of h_1 , it takes $X = [x_1, x_2, x_3]^T$ as input. Moreover, each neuron has another input value, the *bias*, a constant term set to 1 which represents background noise. Thus, together with the weight matrix W_1 , containing one weight w_i^1 for each x_i , another weight b_1^1 is considered, associated to the bias.

¹ <https://www.tensorflow.org/>.

² <https://pytorch.org/>.

³ <https://caffe.berkeleyvision.org/>.

⁴ Generally, a tensor is a n -dimensional array with $n \in \mathbb{N}$.

Fig. 1 Example of a MLP with four layers: the input, two hidden and the output layers. The input $X = [x_1, x_2, x_3]^T$ represents a tensor of size 3×1 . The output $Y = [y_1, y_2, y_3]^T$ represents the 3 possible classes to be assigned to the input, one class for each value in Y

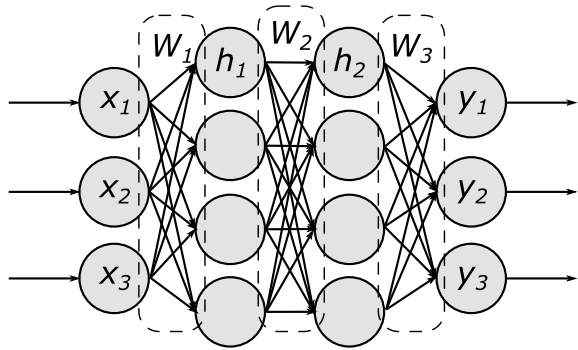
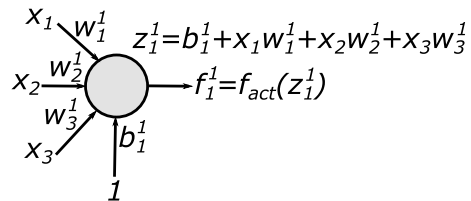


Fig. 2 Computation of neuron h_1 in Fig. 1



This weight is usually automatically added to the neuron by the frameworks used to model the MLP, thus it is not necessary to explicitly add it to the model, as shown in Fig. 1, where the bias terms are not represented.

In detail, first of all, the neuron computes the *network input*, i.e., the input it takes by the network. So, $z_1^1 = b_1^1 + \sum_{i=1}^n x_i w_i^1$ or, in matrix notation $z_1^1 = b_1^1 + X^T W_1$, with $n = 3$ the size of X , i.e., the number of input from the previous layer. Then, z_1^1 is given to the *activation function* f_{act} that computes the output of the neuron, which is given as input to the neurons of the next layer. Thus, $f_1^1(X) = f_{act}(b_1^1 + X^T W_1)$.

There are many possible activation functions, varying from hyperbolic tangent \tanh to the sigmoid or to Rectified Linear Unit (ReLU) [24]. In particular, the last two functions are the most used. Given Z the input of a neuron, the sigmoid function $\sigma(Z) = \frac{1}{1+e^{-Z}}$, shown in Fig. 3a, is used as the activation function of the output layers in case of binary classification. It returns a value between 0 and 1, thus the output layer is designed to have one neuron returning the probability of the input to belong to the positive class (one of the two classes). On the other hand, the ReLU activation function, shown in Fig. 3b, is the most used for the neurons of the hidden layers. It was introduced to improve the performance in terms of training time and defined as $ReLU(H) = \max(0, Z)$.

Therefore, the overall flow is as follow. The input data is multiplied by weights matrix W_1 and given to the neurons of the first layer. Here, every neuron takes the result of this multiplication and applies the activation function, returning the output of the neuron. This value is intended as an indicator of the level of activation of the neuron, the higher the value, the most active that neuron.

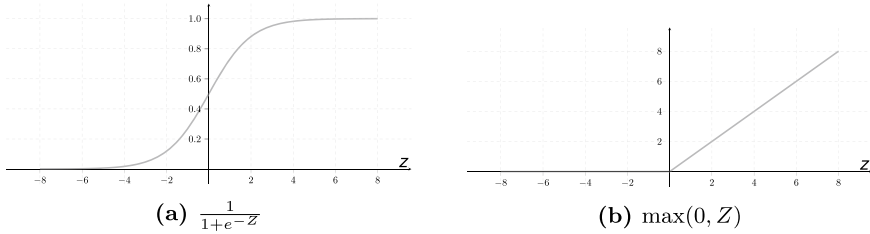


Fig. 3 Sigmoid **a** and ReLU **b** activation functions

This process is then repeated sequentially for the next layers until the output layer is reached. Thus, for example, the output of the neuron h_2 in Fig. 1 is $f_1^2(X) = f_{act}(b_1^2 + Z_1^T W_2)$ where b_1^2 is the bias weight of neuron h_2 , and Z_1 is the tensor containing the output of the neurons of the previous layer.

The last layer will output a probability distribution among the classes considered in the data in the case of classification, or a real value predicting the output label of the example in the case of regression.

During learning, this output distribution is used to compute a loss measure representing how far the predicted output of the network is from the actual label of the input examples. The gradient of this loss measure is calculated w.r.t. the weights. These are then updated by taking a step in the opposite of the direction given by the gradient by means of the gradient descent algorithm or one of its evolutions. This approach is called *back-propagation* and it is at the basis of the training of each neural network, with some variants due to the different architectures adopted.

The need of calculating the gradient guided the search for good activation functions for all the layers. What one would like to have is a function easy to derivate, able to maintain the output values of each neuron in a range that is functional for the training and, at the same time, that does not reduce the information passing through the neuron. Values in the range $[0, 1]$ are the easiest to manage because they avoid an explosion of the weights' values. The counterpart is that multiplying many values lesser than 1 makes the output close to 0, making its gradient close to 0 as well. This problem is called *vanishing gradient*. To try to avoid this problem, the choice of the value of the update step for the weights must be wisely chosen and, in many cases, it must be changed during the training as well.

3 Convolutional Neural Networks

When the input data presents some spatial structure, i.e., each value of the input is connected in some way with other values, MLP may have problems to represent these spatial relations, that usually connect smaller portions of the input example and may appear in different positions and numbers. MLPs need a neuron for each value of the input example, which is a tensor, thus they may require too many neurons

to effectively handle the problem. Input data with spatial structure are, e.g. images, where each pixel is connected with the pixels surrounding it, or time series, where each value is connected with the previous and the following value in the series. If we consider a picture of a person, the set of pixels representing an eye are related and the same relation appears in two different places inside the picture. Moreover, the eye is related to the face of the subject but not, for example, to a car passing behind the person in the background. In these cases, if a MLP is considered, the input layer should have a neuron for each value of the input, i.e., one neuron for each pixel in case of a greyscale image or three neurons for each pixel in case of an RGB picture (one for each channel). This may force the network to have an extremely large number of weights to tune. Consider, for example, a greyscale image with a resolution of 256×256 pixels. The input layer has 65,536 neurons. If the first hidden layer has 100 neurons, which is usually a number too small to be effective given the input size, the network needs $65,536 \cdot 100 = 6,553,600$ weights to connect the two layers. This number will increase even further adding more layers, resulting in a hard to train network.

Convolutional Neural Networks (CNNs) [27] have come to the fore to solve problems where data presents spatial and/or topological structure as in the previous example. CNNs are built using convolutional layers, performing convolutions on the data they receive and extracting *features* from it, typically followed by some fully connected layers to carry out the classification by considering only the extracted features instead of the entire input data. Thus, considering the previous example, the convolutional part checks which features are present in the picture, e.g., if eyes or a mouth or wheels are present. Then, the fully connected part considers the set of features identified in the image and decides which label to assign.

The convolution operation is the integration of two functions x and w , and it is denoted as $x * w$. Function x is called input while function w is the kernel. The result is called feature map, or simply feature. Convolution is defined as

$$(x * w)(t) = \int_{-\infty}^{\infty} x(\tau)w(t - \tau)d\tau$$

or, in the discrete case

$$(x * w)(t) = \sum_{\tau=-\infty}^{\infty} x(\tau)w(t - \tau)$$

In practical scenarios, the input and the kernel functions are tensors, i.e., multidimensional arrays, such as a 2D grid of pixels for an image or a 1D vector for a time series. The kernel scrolls all the input, moving in the directions given by the size of the input itself (in one direction in the case of 1D input, in two directions in the case of 2D input, etc.). Good surveys on the application of CNNs to images and time series are respectively [20, 36].

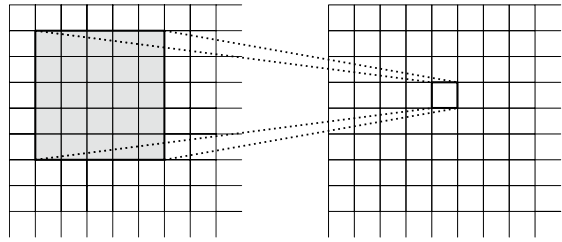
The result of a convolution operation represents the features extracted from the input. The advantage of using CNNs is that the kernel is used on the entire input. The only weights to train are the values of the kernel, therefore, if we consider a single kernel of size 5×5 , the convolutional layer has only 25 weights to train irrespective of the size of the input of the layer. Even if the input has size 256×256 , the number of weights to train will be always 25.

In practice, every convolutional layer applies many kernels at a time, extracting many features from the same image, one for each kernel. For example, one of the first CNN, LeNet5 [26], was defined to train 10 kernels of size 28×28 in the first convolutional layer. Later, Simonyan and Zisserman [42] showed that it is possible to further reduce the number of weights by maintaining the same receptive field by reducing the size of the kernel and adding more convolutional layers, as shown in Fig. 4.

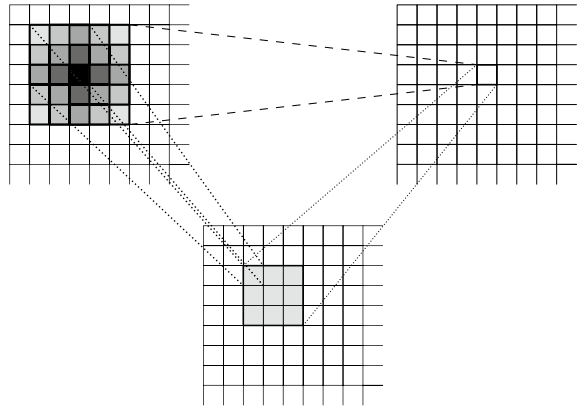
The receptive field is the number of values of the input affecting a single value of the output of a convolution operation. Considering a single convolutional layer having 5×5 kernels, the receptive field of the output of this layer is 25 (Fig. 4a). Consider now replacing this single layer with two convolutional layers having 3 kernels each, as depicted in Fig. 4b. After the second convolutional layer, every value of the output depends on 25 values of the input given to the first convolutional layer, i.e., the receptive field of each value of the output of the second layer is a square 3×3 , so 9 values, of the output of the first layer. In the output of the first layer, each value has a receptive field of 3×3 values of the input. By considering these two layers as a single black-box, the receptive field of the output combines those of the two layers, becoming, as said before, a square of 5×5 values of the input. This is true if we consider a stride equal to 1. The stride indicates by how many pixels the kernel moves in a certain direction to calculate the feature map. The advantage of using more layers is that, while with a single 5×5 convolutional layer there are 25 weights to train, with two 3×3 convolutional layers there are $2 \cdot 9 = 18$ weights to train. If we consider a single layer having a 7×7 kernel and replace it with three layers having 3×3 kernel and using stride 1, the same receptive field is achieved with $3 \cdot 9 = 27$ weights instead of $7 \cdot 7 = 49$. Since every convolutional layer contains several different kernels, the gain in terms of number of weights to train increases fast. By exploiting this approach, it is possible to extract hundreds of features by maintaining the number of weights to train feasible.

However, with the increase of layers, the problem of vanishing gradient will pop up. For this reason, He et al. [16] introduced ResNet where the configuration of layers combines the output of sets of convolutional layers with the input of the first layer in the sets creating a short-circuit between the input and the output of the set of layers, as shown in Fig. 5. This configuration of layers is called *Residual Block* and presents an output function that is $H(X) = F(X) + X$, where $F(X)$ is the output of the set of convolutional layers. Moreover, the increase of complexity of a layer in terms of operations performed also implies an increase of the number of weights. To reduce their number, a possibility is to resort to 1×1 kernels, as in GoogLeNet [46]. Basically, 1×1 convolution is used to reduce the number of features computed by the previous convolutional layer by combining them in a meaningful way. This combi-

Fig. 4 Comparison between 1 convolution operation with kernel 5×5 (a) and 2 convolution operations with kernel 3×3 (b)

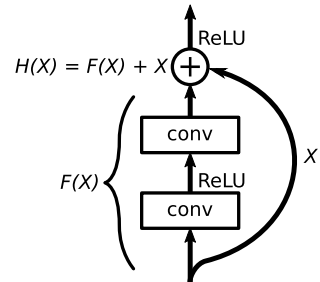


(a) $1 \times \text{conv. } 5 \times 5$.



(b) $2 \times \text{conv. } 3 \times 3$.

Fig. 5 The architecture of a Residual Block as defined in ResNet



nation is learned automatically during the training phase to maintain the information of the features taken as input by the 1×1 convolutional layer while reducing their number. GoogLeNet also introduced the concept of *inception*, i.e., the design of a good local convolutional network architecture, usually with parallel branches, and its use as a new layer.

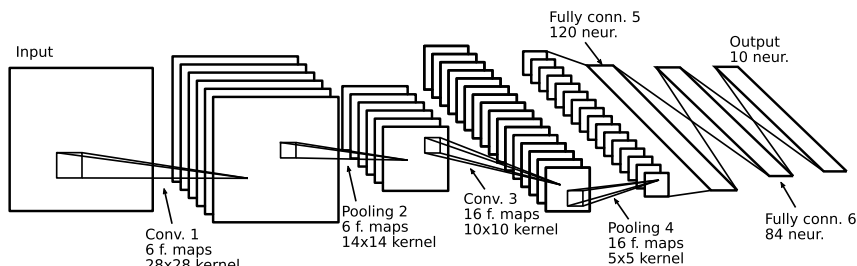
In the last years, many other different architectures have been presented. Most of them combine and extend the ideas presented for GoogLeNet and ResNet by including MLP networks inside layers. An example is the well-known Network-in-Network model [28], combining the idea of residual blocks and inception [49], creating fractal architectures [25], where the model is no more defined by means of layers but by means of a fractal function which is defined recursively, or heavily using 1×1 convolution [19] or short-circuits [18].

Another important operation used by CNNs to reduce feature maps is *pooling*, also called sub-sampling, which usually follows convolution operations. This is used to reduce the size of a feature map by summarizing its information. This summarization can be performed by, e.g., averaging neighbour values or extracting from them the maximum value. The use of convolution and pooling operations allows the extraction of the important features contained in the input data X , representing them by smaller tensors. These features can be used to help classification, or the recognition of certain patterns contained in the data, irrespectively of their position or scale.

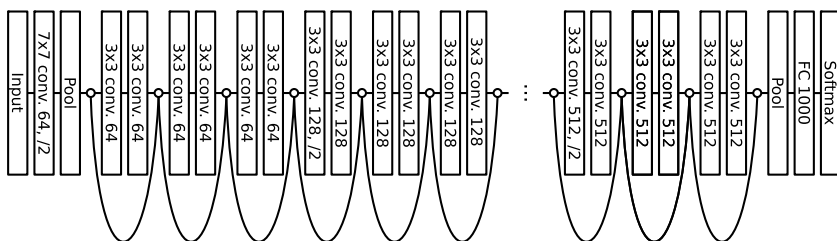
Indeed, CNNs' final layers are usually classical fully connected layers that perform classification on the features extracted by previous layers. Therefore, roughly speaking, a CNN can be divided into two subnetworks, where the first subnetwork processes the input data by means of convolution operations. The output of these operations is then passed to the second subnetwork, which is a fully connected network used to classify the input data. Figure 6 shows the architecture of LeNet5 (Fig. 6a) [26] and of ResNet (Fig. 6b) [16]. From this figure it is possible to see that while the overall architecture has remained the same, i.e., a convolutional part that feeds a set of features to a fully connected part, the depth has increased significantly, going from 6 layers of LeNet5 to 152 layers of ResNet.

Considering images as input, besides the classification task, CNNs are also used to solve more specialized tasks, for example, semantic and instance segmentation, or object detection. The first task consists of identifying which part of the image each pixel belongs to. For example, consider a picture having two dogs in the foreground and a grassland with sky as background, semantic segmentation's objective is that of "classifying" each pixel as sky, grass and dog, while instance segmentation adds the capability of discriminating among the two dogs. Thus, the difference is that semantic segmentation does not discriminate between different subjects, the pixels representing the two dogs are all classified as "dog", while in instance segmentation the pixels representing the first dog are kept separated from those representing the second dog. This task is performed by superimposing a mask on the initial image that colours each pixel depending on how it is classified. This task is usually performed by applying convolution and deconvolution to the image so that the fully connected part of the network is replaced by a sequence of deconvolutional layers [30, 32, 41]. In particular, deconvolution is the dual of convolution, i.e., instead of extracting features from an image, reducing its size, it is aimed at recreating an image starting from a set of features, also possibly increasing its size.

Object detection allows to locate and surround with a box the objects represented in the image. Therefore, given an image, the objective is to identify which objects the image contains, classify them and spatially locate them in the image. This task



(a) LeNet5 [26] executes convolution two times (layers Conv. 1, and Conv. 3), and two pooling operations (Pooling 2, and Pooling 4). Finally, two fully connected layers (5 and 6).



(b) ResNet [16] starts with a convolution and a pooling layer. Then, 148 convolutional layers divided into 74 residual blocks composed of two layers. At the end, one fully connected layer (FC) connects the extracted features to the output layer that uses as activation function the *softmax*, which acts similarly to the sigmoid for multi-class classification.

Fig. 6 The architecture of LeNet5 (a) and ResNet (b)

is performed by defining and training the fully connected layers at the end of the model so that they also output the coordinates of the box highlighting each object. This approach can be used also to estimate the pose of the subject of the image [47], for example to identify if a person is sitting or walking. Object detection and pose estimation can be done by iteratively selecting portions of the image randomly sampled and using this portion as input of the CNN. In this way, the model can extract the information contained in each portion and thus in the entire image [12].

4 Recurrent Neural Networks

If we must process sequential data, the architectures seen before are not the best choice. Sequential data may have different length and be very long, however the model needs to be able to analyse the sequence as a whole. MLPs and CNNs take as input data of fixed size and are designed to handle different characteristics in the

data, such as grid-like topology for the CNNs. When sequential data needs models having a memory of what they have seen previously in the sequence, *Recurrent Neural Networks* (RNNs) [40] may come in handy. An example of an application of the RNNs can be text translation, where given a sentence, the system has to produce a different sentence, which is the translation of the input in a different language. The vanilla RNN is composed of a single neuron that takes as input a single value of the sequence at a time and a value representing its previous state, i.e., the state, usually the output or a function of the output, obtained considering the previous values of the sequence. In a sequence, every individual value represents the value of the sequence at a certain time step t , therefore, the output of a RNN is $h_t(x_t) = f(h_{t-1}, x_t)$ for $t \geq 0$, h_t is the output of the recurrent layer at time t , f the function computed by the layer, and x_t the input at time t . The function f can vary a lot and different types of functions have been defined to manage the process memory in different ways, by replacing the single neuron of the vanilla version with sub-networks containing more neurons. Each version of f uses weights that are learned during the training phase. An important thing to note is that the network is composed by a single neuron/sub-network applying function f , therefore, the weights are tuned by considering the whole sequences, while the network needs less memory than the other types of network discussed in this chapter.

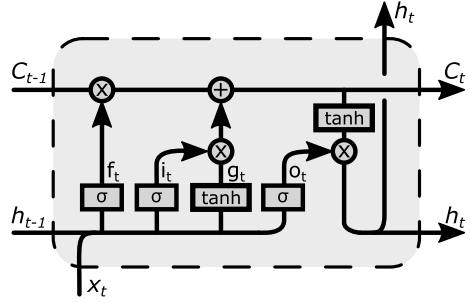
This simple architecture can be used for different purposes, given an input sequence the model could return as output a different sequence or a single value, or else, the model could take a single value as input instead of a sequence and use only its previous state to create a sequence from scratch. This can be easily done by deciding when to return an output, either at each time step, at the end of the input sequence, or every intermediate solution.

The training method used for this type of network is called *back-propagation through time* and in its basic definition is computed for each time step. The difference is that in this case, at each time step, the number of times the weights of the network are considered increases. In fact, at time 1 the output considers the weights of the neuron only once, at time 2 the output is the output of time 1 multiplied by the weights, so they are considered twice, and this holds for every time step until the end of the sequence.

We can also define *Bidirectional-RNN*, where for each time step the output depends also on the future values of the sequence, thus the whole sequence must be read forward and backward in order to return the output. It is worth noting that the possibility of reading the sequence bidirectionally opens to the possibility of taking as input not only sequences but also input that have more dimensions, such as images. This can be done by considering different directions for each dimension, e.g., in the case of an image the network should consider 4 directions.

With the combination of RNNs and CNNs it is also possible to solve the image captioning problem, i.e., given an image, generate a caption describing the content of the image [22]. This can be done by exploiting a CNN trained for solving object recognition task. This network returns a set of labels telling what the image contains. A RNN is then fed with these labels to generate a caption, i.e., a sequence of words, for the image.

Fig. 7 Long Short Term Memory architecture



The main problem of vanilla RNNs is that the network tends to forget the most distant parts of the sequence. Thus, as seen before, more complex architectures have been proposed to improve the memory of the network. One of the most important architecture is called *Long Short Term Memory* RNN (LSTM) [17]. LSTMs replace the single neuron of the vanilla RNN with four gates (basically four neurons) that, for each time step, take as input the current value of the input sequence, and the previous output and state, as shown in Fig. 7. One gate decides which parts of the previous state C_{t-1} to remember by checking the previous output h_{t-1} , another gate computes which parts of the current input x_t to remember, a third gate decides how these two parts are combined together to compute the current state C_t and so to manage what to keep in memory, and the fourth gate computes the current output h_t .

As can be seen in Fig. 7, an LSTM is a network containing several MLP neurons, called gates. To compute the output and the state of the LSTM, it is necessary to compute the output of each gate.

The output of gate f at time t , called *forget* gate, telling what to forget from the previous state is

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

where $\sigma(\cdot)$ is the sigmoid function, W_f and b_f are the weights matrix and the bias weight of gate f , and $[h_{t-1}, x_t]$ is the concatenation of the two tensors h_{t-1} and x_t .

The output of gate i at time t , called *input* gate and telling which values to update, is

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

where W_i and b_i are the weights matrix and the bias weight of gate i .

The output of gate g at time t , called *gate* gate and creating a vector of new candidate values, is

$$g_t = \tanh(W_g \cdot [h_{t-1}, x_t] + b_g)$$

where $\tanh(\cdot)$ is the hyperbolic tangent, W_g and b_g are the weights matrix and the bias weight of gate g .

Now, it is possible to compute C_t as

$$C_t = f_t \times C_{t-1} + i_t \times g_t$$

where \times is the element-wise multiplication.

To find the output h_t , the output of the gate o , called *output gate*, must be computed as:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

where W_o and b_o are the weights matrix and the bias weight of gate o . Finally, the output of the LSTM at time t can be computed as

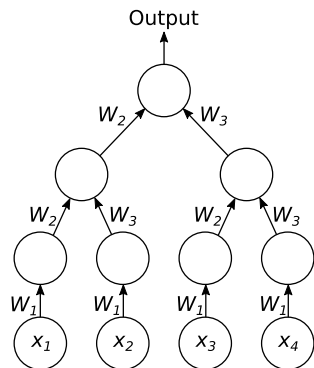
$$h_t = o_t \times \tanh(C_t)$$

Starting from LSTM, a plethora of new models have been proposed. Some of them combine or replace gates to simplify the training, such as the well-known *Gated Recurrent Unit* (GRU) [7], or introduce *spyholes* [11] giving also the previous state as input to (some) of the four gates of a standard LSTM.

The idea of adding sub-networks has also been applied in the case of RNNs [14, 33]. For example, the current state may be passed as input to an MLP whose output is considered as the previous state from the recurrent part [14].

A generalization of RNNs are the *Recursive Neural Networks* [5, 35]. In this case, the model presents a tree-like structure, as shown in Fig. 8. The main advantage is that the number of operations necessary to compute the output given an input sequence is reduced to be logarithmic in the length of the sequence. The drawback is that to achieve the best results the structure of the tree should be tailored to the input. This structure is usually computed by analysing the task. For example, for natural language processing, one can exploit some parsers to get a parse tree to be transformed into the recursive network [44, 45], or define a learner able to automatically create the structure of the tree [5]. However, given the difficulties to correctly identify the best architecture, recursive neural networks are rarely used.

Fig. 8 Recursive neural network



5 Hyper-parameter Optimization

An important aspect to consider for each machine learning approach is the choice of good hyper-parameters. These can be the number of neurons in each layer, the number of layers, the size of the kernels or the number of iterations to perform during training. As seen in the previous sections, with the increasing sophistication of Deep Learning systems, the hyper-parameters and the possible configurations of the network architectures become more and more complex. Furthermore, due to the pervasiveness of Deep Learning systems, the tuning process of the hyper-parameters and the choice of the best neural architecture needs to be addressed even by non-experts.

In the automation of Deep Learning, we can distinguish two families of algorithms: Hyper-Parameters Optimization (HPO) and Neural Architecture Search (NAS) algorithms [9]. With HPO algorithms we modify only the hyper-parameters of the models while with NAS we act only on the architecture of the neural networks.

The four main HPO approaches are Grid Search, Random Search, Bayesian Optimization and with Genetic algorithm. Grid search [51] performs exhaustive search on the specified hyper-parameters space. This algorithm performs a new independent training session for each combination of the hyper-parameters. This algorithm ensures that the optimal configuration is found as long as sufficient resources are provided. This method is guided by a performance metric that decides which of the tested configurations is the best, measuring the performance of the neural model in the training or validation phase. However, due to the fact that the computational resources increase exponentially with the number of hyper-parameters to set, Grid Search suffers from the *curse of dimensionality* [51].

Random Search [3] searches randomly in the user-defined hyper-parameters space. Random search leads to better results than Grid Search, especially when only a small number of hyper-parameters affect the final performance of the learning algorithm [3]. Unlike Grid Search, this algorithm is not guaranteed to achieve the optimum, but may require less computation time while finding a reasonably good model in most cases [3] because the maximum computation time is set before starting the search. With Random Search, it is also possible to include prior knowledge by specifying the distribution from which to sample the hyper-parameter values.

Bayesian Optimization aims to find the global optimum with the minimum number of trials. It is a probabilistic model-based approach for optimizing objective functions which are very expensive or slow to evaluate [8]. Bayesian Optimization builds a probabilistic model (called *surrogate model*) of the objective function and quantifies the uncertainty in this surrogate using a regression model. Then, it uses an acquisition function to decide where to sample the next set of hyper-parameters values [10]. Its effectiveness in optimizing the hyper-parameters of NNs derives from the fact that it limits the number of training sessions by spending more time choosing the next set of hyper-parameters to try. In the literature, there are many examples of application of this kind of HPO to NNs [4, 23, 43].

A genetic algorithm (GA) is a metaheuristic based on Charles Darwin’s evolution theory [50] frequently used to generate high-quality solutions to optimization problems. This algorithm tries to imitate the process of natural selection where the fittest individuals are selected for reproduction to produce the population of the next generation. Evolution starts with a randomly generated population of individuals (each individual is a solution to the optimization problem). One of the key points of GA is the *fitness function*. The fitness function determines the *fitness score* of each individual. Fitness score represents the probability that an individual will be selected for reproduction. At each iteration of the algorithm, with the *selection* phase, the fittest individuals are selected so that they pass on their genes to the next generation. Individuals with high fitness score have more chance to be selected for reproduction. Through the *crossover* phase (also called *recombination*), for each new solution to be produced, a pair of *parent* solution of the actual generation are selected and their genetic information are combined to create new *child* solution. In addition to the crossover, to maintain the genetic diversity, there is also the *mutation*. Mutation alters one or more gene values in an individual from its initial state and occurs during the creation of the new population, according to a user-definable mutation probability. The algorithm terminates where the population has converged (GA does not produce new population that are significantly different from the previous generation).

Then, in the case of DL hyper-parameters optimization, the Genetic algorithm starts with an initial population of N DL models with some predefined hyper-parameters. Then, we can calculate the accuracy (or loss) of the models and uses that as a fitness score. Finally, we can generate a new offspring of DL models. This method is slow (at each iteration, new neural networks are generated which need to be trained) and not guaranteed to find the optimal solution.

NAS is a technique for automating the design of NNs architectures strictly correlated to HPO. NAS methods have already been shown to be capable of overcoming manually designed architectures [37, 52]. The three main elements on which NAS are based are: *Search Space*, *Search Strategy* and *Performance Estimation Strategy* as can be seen in Fig. 9. Search Space refers to all possible architectures that can be generated during the optimization process. Search Strategy refers to the methods for exploring all possible architectures that can be generated by NAS. Performance Estimation Strategy are the methods for measuring the quality of the generated NN [9]. There are different search strategies that can be used to explore the search space of neural architectures. These strategies include: random search, Bayesian Optimiza-

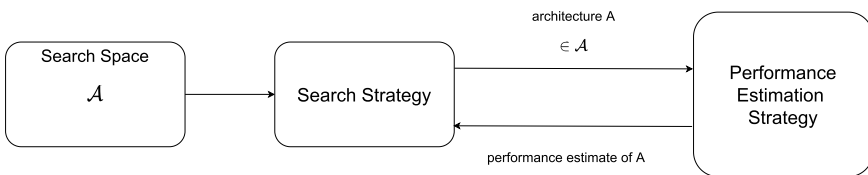


Fig. 9 Components of NAS with their interactions

tion [21], reinforcement learning [34], gradient-based methods [29] and evolutionary algorithms [9, 38]. NAS approaches can be categorized in two groups: classical NAS and one-shot NAS [2]. The first group follows the traditional search approach also used by Grid Search, where each generated NN is trained independently. One-shot NAS algorithms use weight sharing among models in the search space to train a super-net and use this to select better models. This type of algorithms reduces computation resources compared to the classical NAS algorithm. Therefore, a super-net is a single large network that contains every possible operation in the search space. All possible network architecture in the super-net can be considered as a sub-net with shared weights between common edges. Then, rather than training thousands of separate models from scratch like in the classical NAS, one can train a single large network (super-net) capable of emulating any architecture in the search space. Once the super-net is trained, it is used for evaluating the performance of many different architectures sampled at random by zeroing out or removing some operations.

The state-of-the-art of one-shot NAS are: Efficient Neural Architecture Search (ENAS) [34], Differentiable Architecture Search (DARTS) [29], Single Path One-Shot (SPOS) [15] and ProxylessNAS [6]. Nowadays, different software libraries implement this kind of algorithms. We can cite Autokeras⁵ [21] and Neural Network Intelligence (NNI).⁶ Autokeras implements one-shot NAS with Bayesian Optimization-like search strategy and NNI implements both HPO algorithms and classical and one-shot NAS algorithms.

6 Conclusions

This chapter illustrates the main concepts of the Deep Learning field. In particular, it discusses the most important architectures, i.e., Multilayer Perceptrons (MLPs), Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs). While MLPs are a necessary component for performing classification or regression, the other two types are more tailored to specific input types.

CNNs are extremely useful when dealing with input that shows a grid-like topology, such as images and, to some extent, time series. These types of input data are multi-dimensional tensors where each value is connected with the neighbouring values. CNNs owe their success to their capability of extracting features from the input data.

RNNs are well suited for sequential data, such as sentences. Indeed, this kind of architecture is often used for natural language processing, both for reading sentences as input, and generating sentences as output. The main feature of RNNs, absent in CNNs and MLPs, is that they are designed to keep memory of the (most important parts of the) whole input sequence. Therefore, unlike CNNs, RNNs can consider the sequence as a whole instead of considering only a bunch of neighbouring values.

⁵ <https://autokeras.com/>.

⁶ <https://www.microsoft.com/en-us/research/project/neural-network-intelligence/>.

All these architectures can be combined to create powerful tools. For example, to classify images given their content, a CNN needs to send its output to a MLP, which will use the extracted features to perform the classification. Another interesting combination is between CNNs and RNNs to label images. In this case, the extracted features are given as input to a RNN to compose a sentence describing the content of the image.

The architecture of the networks and the algorithms used to train the networks depend on many hyper-parameters, that must be optimized to obtain good results. Since the number of these hyper-parameters can be high, we need mechanisms to automatically tune their values. For this reason, in this chapter we have surveyed the relevant literature about hyper-parameters' optimization.

References

1. C.C. Aggarwal, *Neural Networks and Deep Learning—A Textbook* (Springer, 2018)
2. G. Bender, P. Kindermans, B. Zoph, V. Vasudevan, Q.V. Le, Understanding and simplifying one-shot architecture search, in *Proceedings of the 35th International Conference on Machine Learning, ICML 2018*, Stockholm, Sweden, 10–15 July 2018, eds. by J.G. Dy, A. Krause. Proceedings of Machine Learning Research, vol. 80, pp. 549–558. (PMLR, 2018), <http://proceedings.mlr.press/v80/bender18a.html>
3. J. Bergstra, Y. Bengio, Random search for hyper-parameter optimization. *J. Mach. Learn. Res.* **13**, 281–305 (2012). <http://dl.acm.org/citation.cfm?id=2188395>
4. H. Bertrand, R. Ardon, M. Perrot, I. Bloch, Hyperparameter optimization of deep neural networks: combining hyperband with bayesian model selection, in *Conférence sur l'Apprentissage Automatique* (2017)
5. L. Bottou, From machine learning to machine reasoning—an essay. *Mach. Learn.* **94**(2), 133–149 (2014)
6. H. Cai, L. Zhu, S. Han, Proxylessnas: direct neural architecture search on target task and hardware. arXiv preprint [arXiv:1812.00332](https://arxiv.org/abs/1812.00332) (2018)
7. K. Cho, B. van Merriënboer, D. Bahdanau, Y. Bengio, On the properties of neural machine translation: encoder-decoder approaches, in *Proceedings of SSST@EMNLP 2014, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*, eds by D. Wu, M. Carpuat, X. Carreras, E.M. Vecchi (Doha, Qatar, 2014), pp. 103–111. Association for Computational Linguistics (2014). <https://doi.org/10.3115/v1/W14-4012>, <https://www.aclweb.org/anthology/W14-4012/>
8. I. Dewancker, M. McCourt, S. Clark, Bayesian optimization primer (2015), https://app.sigopt.com/static/pdf/SigOpt_Bayesian_Optimization_Primer.pdf
9. T. Elsken, J.H. Metzen, F. Hutter, Neural architecture search: a survey. arXiv preprint [arXiv:1808.05377](https://arxiv.org/abs/1808.05377) (2018)
10. P.I. Frazier, A tutorial on bayesian optimization. arXiv preprint [arXiv:1807.02811](https://arxiv.org/abs/1807.02811) (2018)
11. F.A. Gers, J. Schmidhuber, Recurrent nets that time and count, in *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks, IJCNN 2000, Neural Computing: New Challenges and Perspectives for the New Millennium*, Vol. 3, pp. 189–194. Como, Italy, 24–27 July 2000. IEEE Computer Society (2000). <https://doi.org/10.1109/IJCNN.2000.861302>
12. R.B. Girshick, J. Donahue, T. Darrell, J. Malik, Rich feature hierarchies for accurate object detection and semantic segmentation, in *2014 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2014*, pp. 580–587. Columbus, OH, USA, 23–28 June 2014. IEEE Computer Society (2014). <https://doi.org/10.1109/CVPR.2014.81>

13. I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning*, vol. 1 (MIT Press, 2016)
14. A. Graves, A. Mohamed, G.E. Hinton, Speech recognition with deep recurrent neural networks, in *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2013*, pp. 6645–6649. Vancouver, BC, Canada, 26–31 May 2013. IEEE Press (2013). <https://doi.org/10.1109/ICASSP.2013.6638947>
15. Z. Guo, X. Zhang, H. Mu, W. Heng, Z. Liu, Y. Wei, J. Sun, Single path one-shot neural architecture search with uniform sampling. arXiv preprint [arXiv:1904.00420](https://arxiv.org/abs/1904.00420) (2019)
16. K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016*, pp. 770–778. Las Vegas, NV, USA, June 27–30 2016. IEEE Computer Society (2016). <https://doi.org/10.1109/CVPR.2016.90>
17. S. Hochreiter, J. Schmidhuber, Long short-term memory. *Neural Comput.* **9**(8), 1735–1780 (1997)
18. G. Huang, Z. Liu, L. van der Maaten, K.Q. Weinberger, Densely connected convolutional networks, in *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, pp. 2261–2269. Honolulu, HI, USA, 21–26 July 2017. IEEE Computer Society (2017). <https://doi.org/10.1109/CVPR.2017.243>
19. F.N. Iandola, M.W. Moskewicz, K. Ashraf, S. Han, W.J. Dally, K. Keutzer, SqueezeNet: alexnet-level accuracy with 50x fewer parameters and <1mb model size. *CoRR* **abs/1602.07360** (2016), <http://arxiv.org/abs/1602.07360>
20. H. Ismail Fawaz, G. Forestier, J. Weber, L. Idoumghar, P.A. Muller, Deep learning for time series classification: a review. *Data Mining Knowl. Discov.* **33**(4), 917–963 (2019)
21. H. Jin, Q. Song, X. Hu, Auto-keras: an efficient neural architecture search system, in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1946–1956 (2019)
22. A. Karpathy, L. Fei-Fei, Deep visual-semantic alignments for generating image descriptions. *IEEE Trans. Pattern Analysis Mach. Intell.* **39**(4), 664–676 (2017), <https://doi.org/10.1109/TPAMI.2016.2598339>
23. R. Korichi, M. Guillemot, C. Heusèle, Tuning neural network hyperparameters through bayesian optimization and application to cosmetic formulation data, in *ORASIS 2019* (2019)
24. A. Krizhevsky, I. Sutskever, G.E. Hinton, Imagenet classification with deep convolutional neural networks, in *26th Annual Conference on Neural Information Processing Systems 2012*, eds by P.L. Bartlett, F.C.N. Pereira, C.J.C. Burges, L. Bottou, K.Q. Weinberger, pp. 1106–1114. Lake Tahoe, Nevada, United States, 3–6 Dec 2012. *Advances in Neural Information Processing Systems 25*. <http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks>
25. G. Larsson, M. Maire, G. Shakhnarovich, Fractalnet: ultra-deep neural networks without residuals, in *International Conference on Learning Representations*. OpenReview.net (2017) <https://openreview.net/forum?id=S1VaB4cex>
26. Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, Gradient-based learning applied to document recognition. *Proc. IEEE* **86**(11), 2278–2324 (1998)
27. Y. LeCun et al., Generalization and network design strategies. *Connect. Perspect.* **19**, 143–155 (1989)
28. M. Lin, Q. Chen, S. Yan, Network in network, in *2nd International Conference on Learning Representations, ICLR 2014* eds by Y. Bengio, Y. LeCun (eds.), Banff, AB, Canada, 14–16 April 2014. *Conference Track Proceedings* (2014), <http://arxiv.org/abs/1312.4400>
29. H. Liu, K. Simonyan, Y. Yang, Darts: differentiable architecture search. arXiv preprint [arXiv:1806.09055](https://arxiv.org/abs/1806.09055) (2018)
30. J. Long, E. Shelhamer, T. Darrell, Fully convolutional networks for semantic segmentation, in *2015 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015*, pp. 3431–3440. Boston, MA, USA, 7–12 June 2015. IEEE Computer Society (2015). <https://doi.org/10.1109/CVPR.2015.7298965>
31. W.S. McCulloch, W. Pitts, A logical calculus of the ideas immanent in nervous activity. *Bull. Math. Biophys.* **5**(4), 115–133 (1943)

32. H. Noh, S. Hong, B. Han, Learning deconvolution network for semantic segmentation, in *2015 IEEE International Conference on Computer Vision, ICCV 2015*, pp. 1520–1528. Santiago, Chile, 7–13 December 2015. IEEE Computer Society (2015). <https://doi.org/10.1109/ICCV.2015.178>
33. R. Pascanu, C. Gülçehre, K. Cho, Y. Bengio, How to construct deep recurrent neural networks, in *2nd International Conference on Learning Representations, ICLR 2014*, eds by Y. Bengio, Y. LeCun. Banff, AB, Canada, 14–16 April 2014. Conference Track Proceedings (2014). <http://arxiv.org/abs/1312.6026>
34. H. Pham, M.Y. Guan, B. Zoph, Q.V. Le, J. Dean, Efficient neural architecture search via parameter sharing. arXiv preprint [arXiv:1802.03268](https://arxiv.org/abs/1802.03268) (2018)
35. J.B. Pollack, Recursive distributed representations. *Artif. Intell.* **46**(1–2), 77–105 (1990)
36. W. Rawat, Z. Wang, Deep convolutional neural networks for image classification: a comprehensive review. *Neural Comput.* **29**, 1–98 (2017)
37. E. Real, A. Aggarwal, Y. Huang, Q.V. Le, Aging evolution for image classifier architecture search, in *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence (AAAI-19)*. (AAAI Press/IJCAI, 2019)
38. E. Real, A. Aggarwal, Y. Huang, Q.V. Le, Regularized evolution for image classifier architecture search, in *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence (AAAI-19)*. vol. 33, pp. 4780–4789. AAAI Press/IJCAI (2019)
39. F. Rosenblatt, The perceptron: a probabilistic model for information storage and organization in the brain. *Psychol. Rev.* **65**(6), 386 (1958)
40. D.E. Rumelhart, G.E. Hinton, R.J. Williams, Learning representations by back-propagating errors. *Nature* **323**(6088), 533–536 (1986)
41. E. Shelhamer, J. Long, T. Darrell, Fully convolutional networks for semantic segmentation. *IEEE Trans. Pattern Analysis Mach. Intell.* **39**(4), 640–651 (2017)
42. K. Simonyan, A. Zisserman, Very deep convolutional networks for large-scale image recognition, in *3rd International Conference on Learning Representations, ICLR 2015*, eds by Y. Bengio, Y. LeCun. San Diego, CA, USA, 7–9 May 2015, Conference Track Proceedings (2015). <http://arxiv.org/abs/1409.1556>
43. J. Snoek, H. Larochelle, R.P. Adams, Practical bayesian optimization of machine learning algorithms, in *26th Annual Conference on Neural Information Processing Systems 2012* eds by P.L. Bartlett, F.C.N. Pereira, C.J.C. Burges, L. Bottou, K.Q. Weinberger, pp. 2960–2968. Advances in Neural Information Processing Systems 25. Proceedings of a meeting held 3–6 December 2012, Lake Tahoe, Nevada, United States (2012). <http://papers.nips.cc/paper/4522-practical-bayesian-optimization-of-machine-learning-algorithms>
44. R. Socher, E.H. Huang, J. Pennington, A.Y. Ng, C.D. Manning, Dynamic pooling and unfolding recursive autoencoders for paraphrase detection, in *25th Annual Conference on Neural Information Processing Systems 2011*, eds by J. Shawe-Taylor, R.S. Zemel, P.L. Bartlett, F.C.N. Pereira, K.Q. Weinberger, pp. 801–809. Advances in Neural Information Processing Systems 24. Proceedings of a meeting held 12–14 December 2011, Granada, Spain (2011). <http://papers.nips.cc/paper/4204-dynamic-pooling-and-unfolding-recursive-autoencoders-for-paraphrase-detection>
45. R. Socher, A. Perelygin, J. Wu, J. Chuang, C.D. Manning, A.Y. Ng, C. Potts, Recursive deep models for semantic compositionality over a sentiment treebank, in *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing, EMNLP 2013*, pp. 1631–1642. Grand Hyatt Seattle, Seattle, Washington, USA, 18–21 October 2013. A meeting of SIGDAT, a Special Interest Group of the ACL. ACL (2013). <https://www.aclweb.org/anthology/D13-1170/>
46. C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S.E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, Going deeper with convolutions, in *2015 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015*, pp. 1–9. Boston, MA, USA, 7–12 June 2015. IEEE Computer Society (2015). <https://doi.org/10.1109/CVPR.2015.7298594>

47. A. Toshev, C. Szegedy, Deeppose: human pose estimation via deep neural networks, in *2014 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2014*, . pp. 1653–1660. Columbus, OH, USA, 23–28 June 2014. IEEE Computer Society (2014). <https://doi.org/10.1109/CVPR.2014.214>
48. B. Widrow, M.E. Hoff, *Adaptive Switching Circuits* (Stanford Univ Ca Stanford Electronics Labs, Tech. rep., 1960)
49. S. Xie, R.B. Girshick, P. Dollár, P., Tu, Z., He, K.: Aggregated residual transformations for deep neural networks, in *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, pp. 5987–5995. Honolulu, HI, USA, 21–26 July 2017. IEEE Computer Society (2017). <https://doi.org/10.1109/CVPR.2017.634>
50. X.S. Yang, Chapter 6—genetic algorithms, in *Nature-Inspired Optimization Algorithms*, 2nd edn, eds by X.S. Yang, pp. 91–100. (Academic Press, 2021). <https://doi.org/10.1016/B978-0-12-821986-7.00013-5>, <https://www.sciencedirect.com/science/article/pii/B9780128219867000135>
51. T. Yu, H. Zhu, Hyper-parameter optimization: a review of algorithms and applications. arXiv preprint [arXiv:2003.05689](https://arxiv.org/abs/2003.05689) (2020)
52. B. Zoph, V. Vasudevan, J. Shlens, Q.V. Le, Learning transferable architectures for scalable image recognition, in *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018*, pp. 8697–8710. Salt Lake City, UT, USA, 18–22 June 2018. IEEE Computer Society (2018). <https://doi.org/10.1109/CVPR.2018.00907>