Universidade Presbiteriana Mackenzie

Princípios de Deep Learning

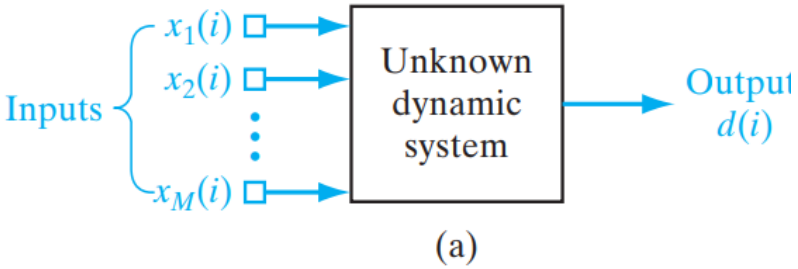
Prof. Dr. Leandro Augusto da Silva
leandroaugusto.silva@mackenzie.br

Laboratório de Big Data e Métodos Analíticos Aplicados

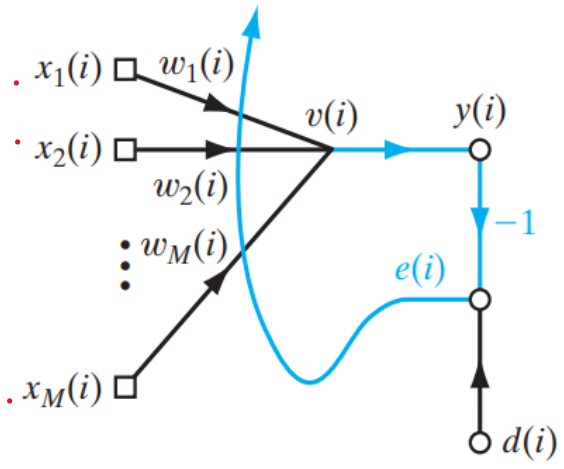
Faculdade de Computação e Informática
Programa de Pós-Graduação em Engenharia Elétrica e Computação
Programa de Pós-Graduação em Computação Aplicada

O Problema de Filtragem Adaptativa

FIGURE 3.1 (a) Unknown dynamic system. (b) Signal-flow graph of adaptive model for the system; the graph embodies a feedback loop set in color.



(a)



(b)

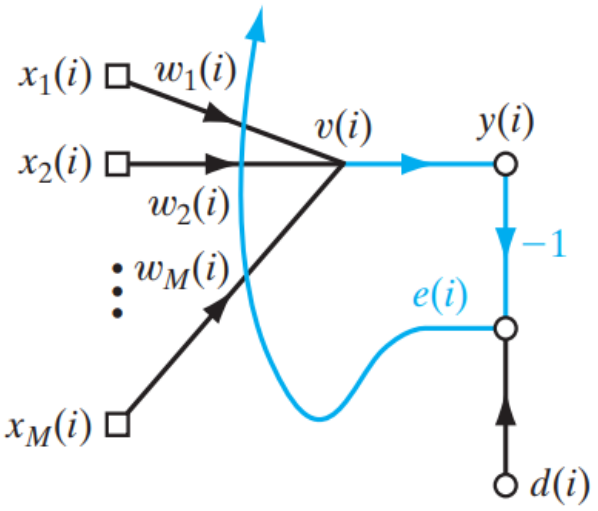
O Problema de Filtragem Adaptativa

- O problema é o de como projetar um modelo de múltiplas entradas-única saída do sistema dinâmico desconhecido.
- O modelo neuronal opera sob a influência de um algoritmo que controla os ajustes necessários dos pesos sinápticos do neurônio, considerando:
  - O algoritmo inicia com uma configuração arbitrária de pesos
  - Os ajustes são feitos de forma contínua
  - Os cálculos dos ajustes são completados dentro de um intervalo de tempo

O Problema de Filtragem Adaptativa

- O modelo neuronal descrito é conhecido como um filtro adaptativo, cuja operação é constituída de dois processos:

  1) Processo de Filtragem
     - Uma saída: y(i)
     - Um sinal de erro
  2) Processo adaptativo
     - Ajuste dos pesos de acordo com o sinal de erro



Técnicas de otimização irrestritas

- A maneira pela qual o sinal de erro é usado para controlar os ajustes dos pesos é determinada pela *função de custo (*$\mathscr{E}(\mathbf{w})$*)*
- Considere a função diferenciável de um vetor de peso (parâmetro) desconhecido $\mathbf{w}$. O objetivo é encontrar a solução ótima $\mathbf{w}^*$

$$\mathscr{E}(\mathbf{w}^*) \leq \mathscr{E}(\mathbf{w})$$

- Ou seja, deseja-se minimizar $\mathscr{E}(\mathbf{w})$ em relação a $\mathbf{w}$

Técnicas de otimização irrestritas

A condição necessária para a otimização é

$$\nabla \mathscr{E}(\mathbf{w}^*) = \mathbf{0} \tag{3.7}$$

onde ∇ é o *operador gradiente*:

$$\nabla = \left[\frac{\partial}{\partial w_1}, \frac{\partial}{\partial w_2}, \cdots, \frac{\partial}{\partial w_m} \right]^T \tag{3.8}$$

e $\nabla \mathscr{E}(\mathbf{w})$ é o *vetor gradiente* da função de custo:

$$\nabla \mathscr{E}(\mathbf{w}) = \left[\frac{\partial \mathscr{E}}{\partial w_1}, \frac{\partial \mathscr{E}}{\partial w_2}, \cdots, \frac{\partial \mathscr{E}}{\partial w_m} \right]^T \tag{3.9}$$

Uma classe de algoritmos de otimização irrestritos que é particularmente adequada para o projeto de filtros adaptativos é baseada na idéia da *descida iterativa* local:

> *Iniciando com uma suposição inicial representada por* $\mathbf{w}(0)$, *gere uma seqüência de vetores de peso* $\mathbf{w}(1), \mathbf{w}(2), \ldots$, *de modo que a função de custo* $\mathscr{E}(\mathbf{w})$ *seja reduzida a cada iteração do algoritmo, como mostrado por*

$$\mathscr{E}(\mathbf{w}(n+1)) < \mathscr{E}(\mathbf{w}(n)) \tag{3.10}$$

> *onde* $\mathbf{w}(n)$ *é o valor antigo do vetor de peso e* $\mathbf{w}(n+1)$ *é o seu valor atualizado.*

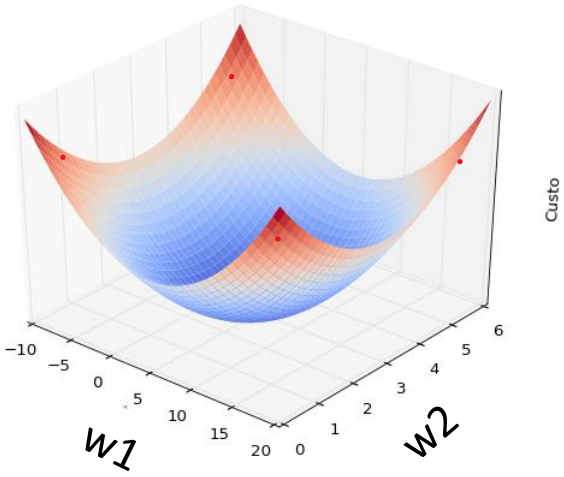
Esperamos que este algoritmo eventualmente convirja para a solução ótima $\mathbf{w}^*$. Dizemos "esperamos" porque há uma nítida possibilidade de o algoritmo divergir (i.e., se tornar instável) a menos que sejam tomadas precauções especiais.

Haykin (2001_, pg 147)

Gradiente descendente*

- Para visualização da função de custo, considere um neurônio com duas entradas, portanto dois pesos plotados com a função de custo
- O ponto de mínimo está onde w_1=5 e w_2=3. Portanto:

$$\nabla = \left[\frac{\partial}{\partial w1}, \frac{\partial}{\partial w2}\right]$$



- O gradiente é simplesmente um vetor de derivadas parciais que dão a inclinação da "tigela" em cada ponto e em cada direção
- Portanto, a direção oposta ao gradiente leva ao ponto de mínimo

*https://matheusfacure.github.io/2017/02/20/MQO-Gradiente-Descendente/

Método de Descida mais íngrime

No método da descida mais íngreme, os ajustes sucessivos aplicados ao vetor de peso $\mathbf{w}$ são na direção da descida mais íngreme, isto é, em uma direção oposta ao *vetor do gradiente* $\nabla\mathscr{E}(\mathbf{w})$. Por conveniência de apresentação, escrevemos

$$\mathbf{g} = \nabla\mathscr{E}(\mathbf{w}) \tag{3.11}$$

Correspondentemente, o algoritmo da descida mais íngreme é descrito formalmente por

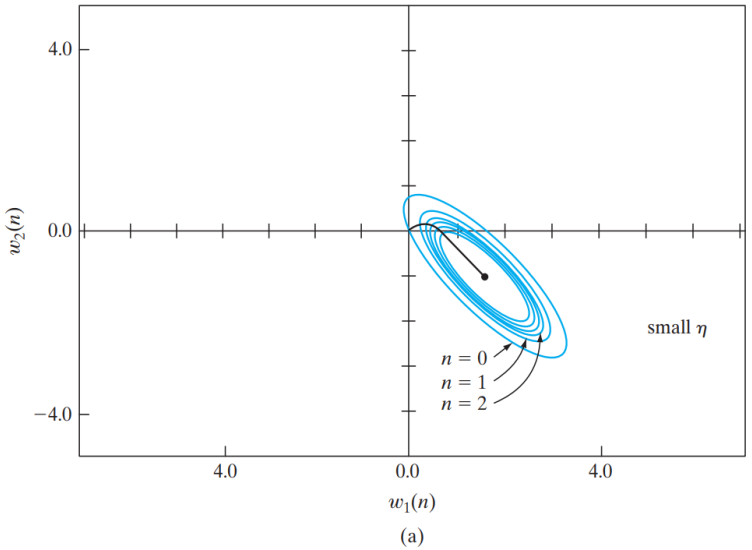
$$\mathbf{w}(n+1) = \mathbf{w}(n) - \eta\mathbf{g}(n) \tag{3.12}$$

onde η é uma constante positiva chamada de *tamanho do passo* ou *parâmetro de taxa de aprendizagem*, e $\mathbf{g}(n)$ é o vetor do gradiente calculado no ponto $\mathbf{w}(n)$. Passando da iteração n para $n+1$, o algoritmo aplica a *correção*

$$\begin{aligned}\Delta\mathbf{w}(n) &= \mathbf{w}(n+1) - \mathbf{w}(n) \\ &= -\eta\mathbf{g}(n)\end{aligned} \tag{3.13}$$

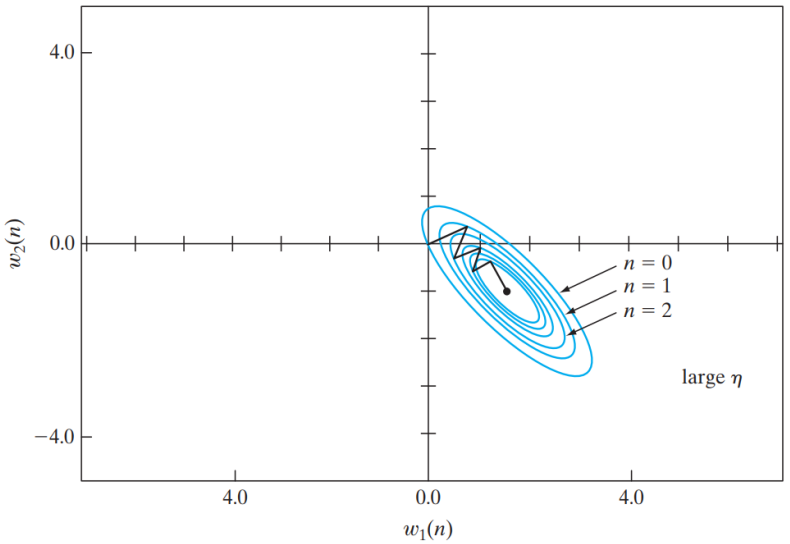
$$\mathbf{g}(n) = -\, e \times \mathbf{x}(n)$$

Método de Descida mais íngrime



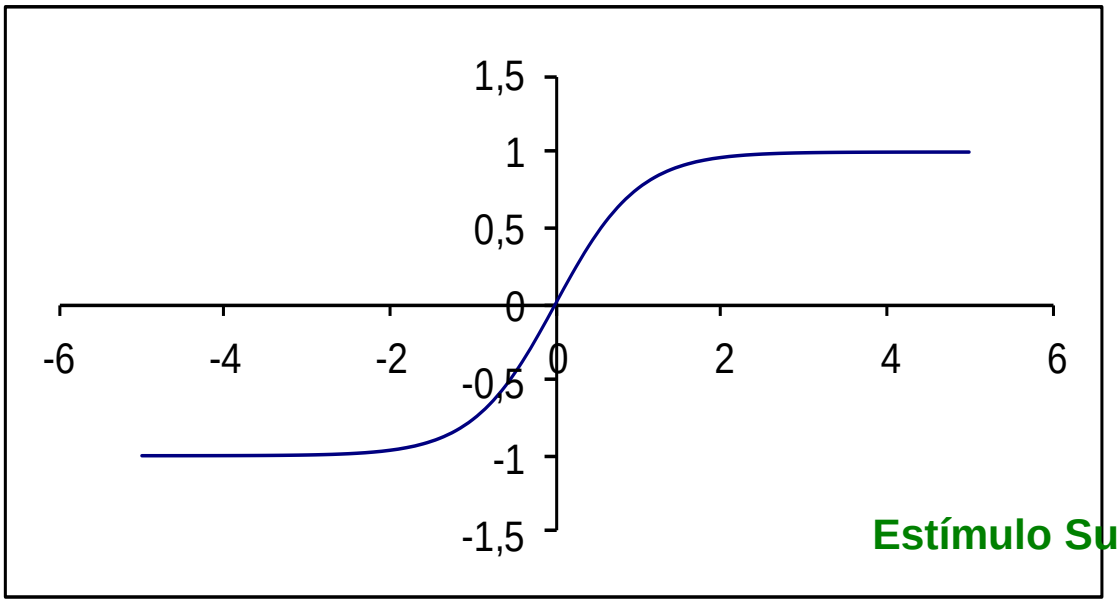
(a)

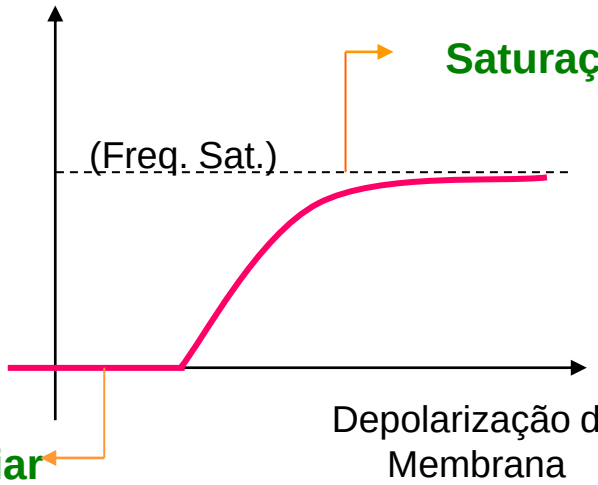
Quando η é pequeno, a resposta transitória do algoritmo é *sobreamortecida*, sendo que a trajetória traçada por **w**(*n*) segue um caminho suave no plano *W*, como ilustrado na Fig. 3.2 a.



Quando η é grande, a resposta transitória do algoritmo é *subamortecida*, sendo que a trajetória de **w**(*n*) segue um caminho ziguezagueante (oscilatório), como ilustrado na Fig. 3.2 b.

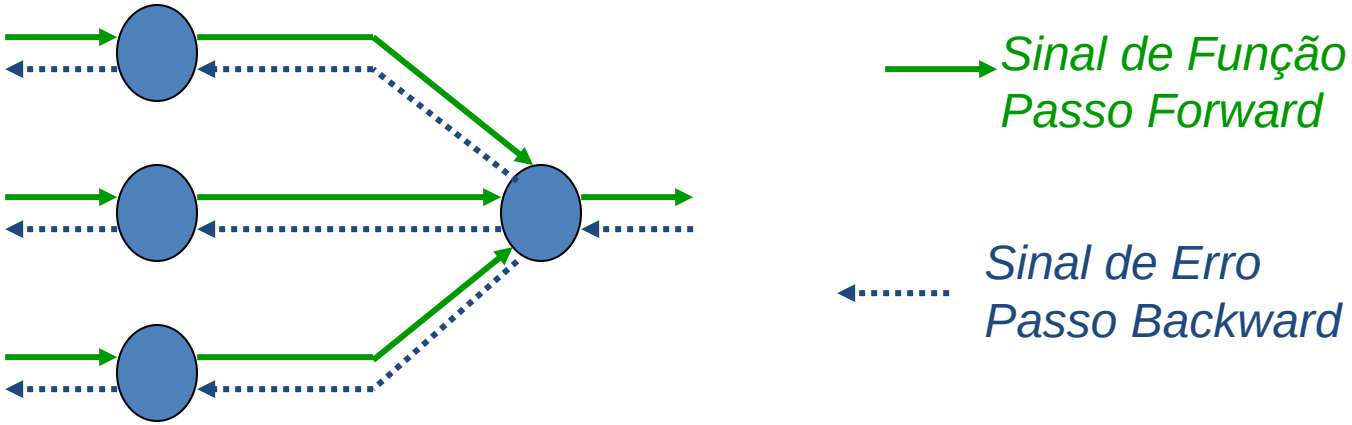
Função de transferência





Algoritmo de Aprendizado

- Algoritmo Back-propagation



- Os pesos da rede são ajustados com o objetivo de minimizar o erro médio quadrático.

Erro Médio Quadrático

- O sinal de erro da saída do neurônio ***j*** na apresentação dos ***n-th*** exemplos de treinamento:

- Erro na saída do neurônio:

$$e_j(n) = d_j(n) - y_j(n)$$

- Energia total do erro:

$$E(n) = \frac{1}{2}\sum_{j \in C} e_j^2(n)$$

C: conjunto de neurônios na camada de saída

- Energia média do erro quadrático:

$$E_{AV} = \frac{1}{N}\sum_{n=1}^{N} E(n)$$

N: tamanho do conjunto de treinamento

- **Objetivo:** ***Ajustar os pesos da rede para diminuir E_{AV}***

Notação

e_j Erro na saída do neurônio j

y_j Saída do neurônio j

$$v_j = \sum_{i=0,\ldots,m} w_{ji} y_i$$

Campo local induzido do neurônio j

Erro e_j do neurônio de saída

- Caso 1: ***j neurônio de saída***

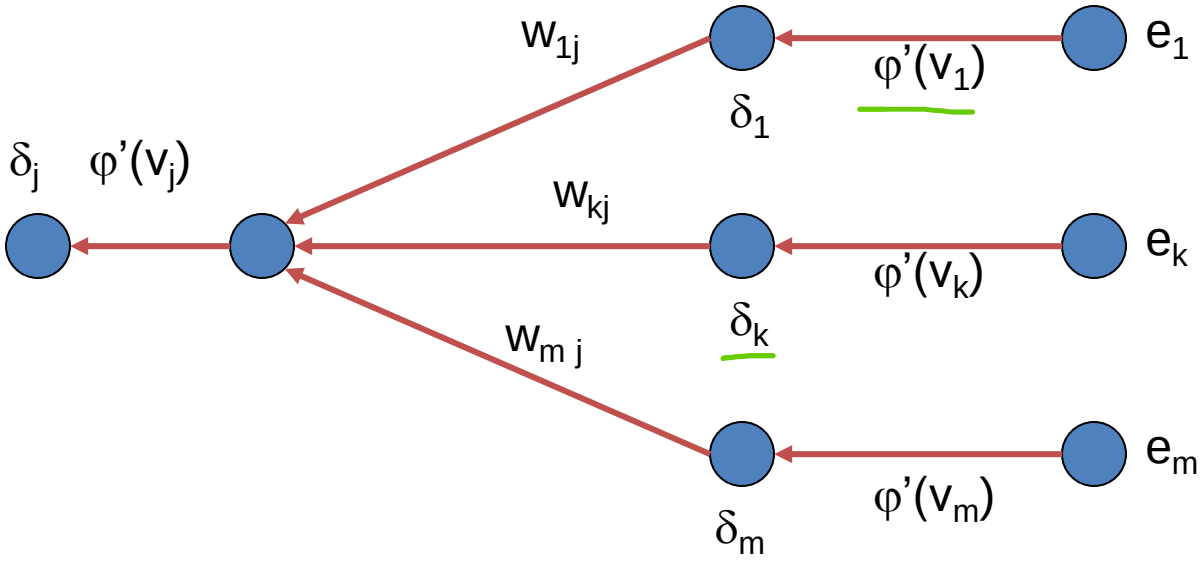
$$e_j = d_j - y_j$$

Então

$$\delta_j = (d_j - y_j)\varphi'(v_j)$$

Gradiente Local do Neurônio Escondido

$$\delta_j = \varphi'(v_j) \sum_{k \in C} \delta_k w_{kj}$$



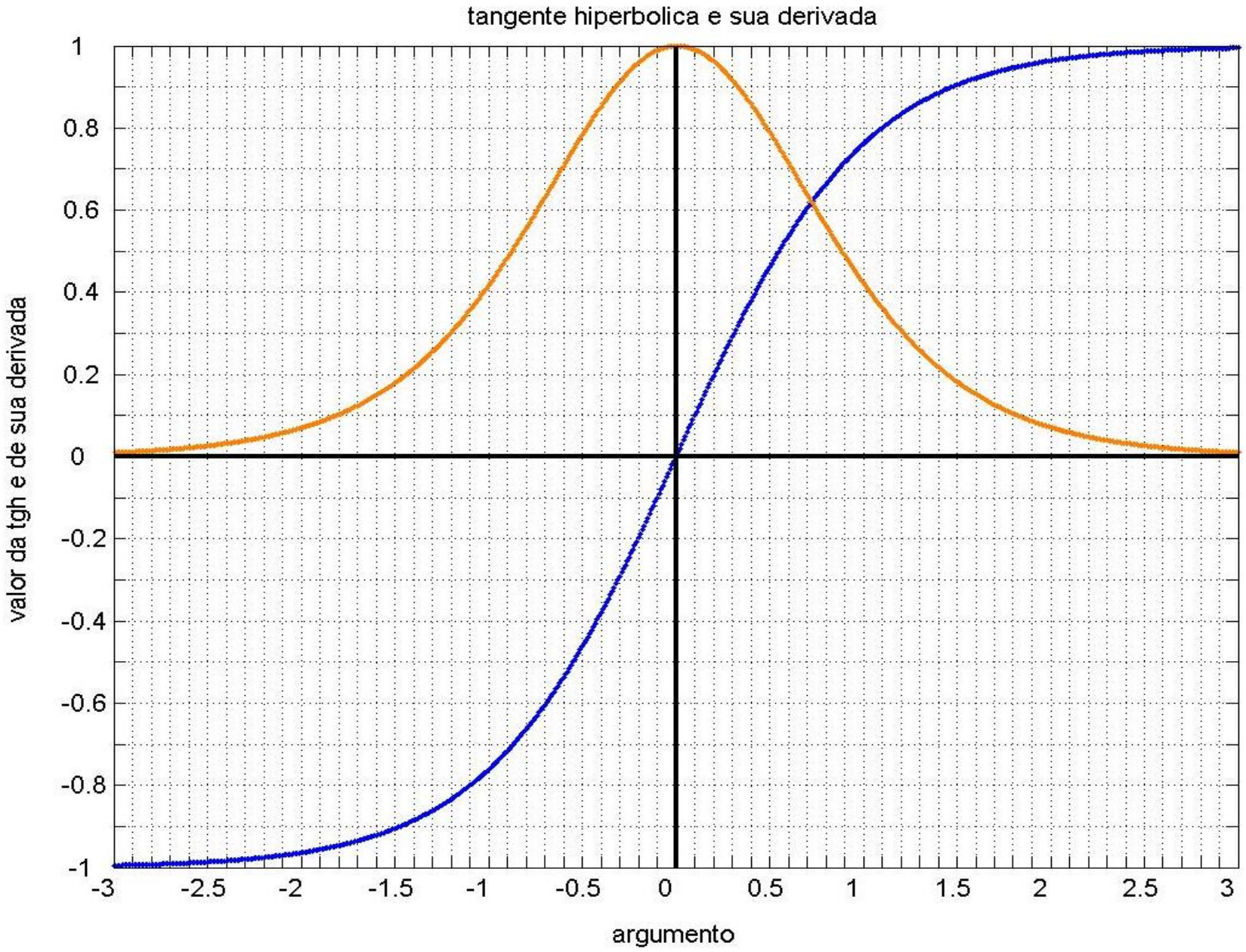
Grafo orientado do sinal do erro retropropagado neurônio ***j***

Regra Delta

- **Regra Delta** $\Delta w_{ji} = \eta \delta_j y_i$

$$\delta_j = \begin{cases} \varphi'(v_j)(d_j - y_j) & \text{SE j nó de saída} \\ \varphi'(v_j)\sum_{k \in C} \delta_k w_{kj} & \text{SE j nó escondido} \end{cases}$$

C: Conjunto de neurônios na camada que contém um ***j***


tangente hiperbolica e sua derivada
valor da tgh e de sua derivada
argumento
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-3
-2.5
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2
2.5
3

Deep Neural Net

In Chapter 10 we introduced artificial neural networks and trained our first deep neural network. But it was a very shallow DNN, with only two hidden layers. What if you need to tackle a very complex problem, such as detecting hundreds of types of objects in high-resolution images? You may need to train a much deeper DNN, perhaps with (say) 10 layers, each containing hundreds of neurons, connected by hundreds of thousands of connections. This would not be a walk in the park:

- First, you would be faced with the tricky *vanishing gradients* problem (or the related *exploding gradients* problem) that affects deep neural networks and makes lower layers very hard to train.
- Second, with such a large network, training would be extremely slow.
- Third, a model with millions of parameters would severely risk overfitting the training set.

In this chapter, we will go through each of these problems in turn and present techniques to solve them. We will start by explaining the vanishing gradients problem and exploring some of the most popular solutions to this problem. Next we will look at various optimizers that can speed up training large models tremendously compared to plain Gradient Descent. Finally, we will go through a few popular regularization techniques for large neural networks.

With these tools, you will be able to train very deep nets: welcome to Deep Learning!

Vanishing Gradient Problems

> Unfortunately, gradients often get smaller and smaller as the algorithm progresses down to the lower layers. As a result, the Gradient Descent update leaves the lower layer connection weights virtually unchanged, and training never converges to a good solution. This is called the *vanishing gradients* problem. In some cases, the opposite can happen: the gradients can grow bigger and bigger, so many layers get insanely large weight updates and the algorithm diverges. This is the *exploding gradients* problem, which is mostly encountered in recurrent neural networks (see Chapter 14). More generally, deep neural networks suffer from unstable gradients; different layers may learn at widely different speeds.

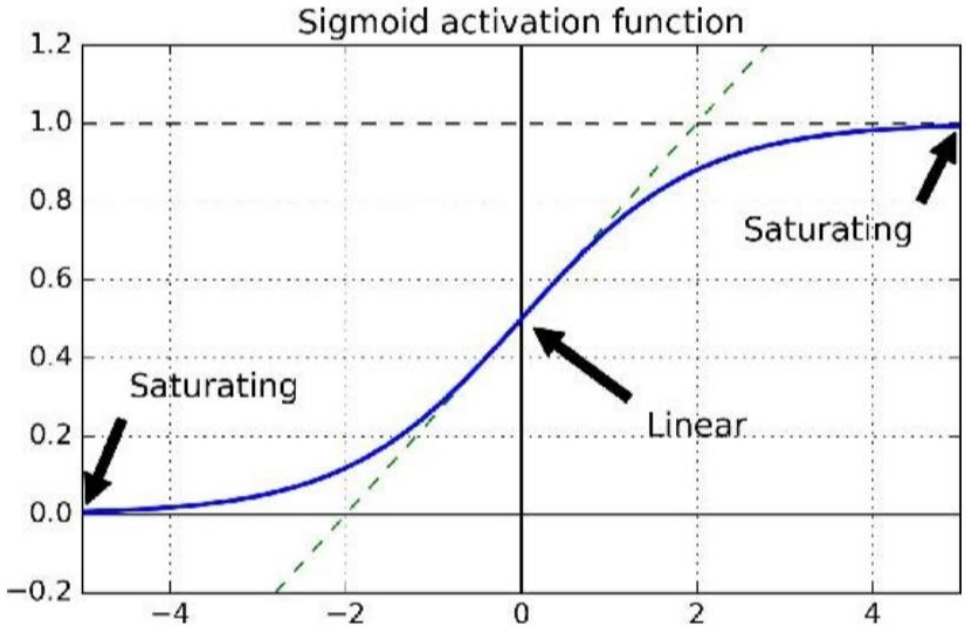


Figure 11-1. Logistic activation function saturation

Looking at the logistic activation function (see Figure 11-1), you can see that when inputs become large (negative or positive), the function saturates at 0 or 1, with a derivative extremely close to 0. Thus when backpropagation kicks in, it has virtually no gradient to propagate back through the network, and what little gradient exists keeps getting diluted as backpropagation progresses down through the top layers, so there is really nothing left for the lower layers.

Xavier and He Initialization

In their paper, Glorot and Bengio propose a way to significantly alleviate this problem. We need the signal to flow properly in both directions: in the forward direction when making predictions, and in the reverse direction when backpropagating gradients. We don't want the signal to die out, nor do we want it to explode and saturate. For the signal to flow properly, the authors argue that we need the variance of the outputs of each layer to be equal to the variance of its inputs,[2] and we also need the gradients to have equal variance before and after flowing through a layer in the reverse direction (please check out the paper if you are interested in the mathematical details). It is actually not possible to guarantee both unless the layer has an equal number of input and output connections, but they proposed a good compromise that has proven to work very well in practice: the connection weights must be initialized randomly as

Nonsaturating Activation Functions

One of the insights in the 2010 paper by Glorot and Bengio was that the vanishing/exploding gradients problems were in part due to a poor choice of activation function. Until then most people had assumed that if Mother Nature had chosen to use roughly sigmoid activation functions in biological neurons, they must be an excellent choice. But it turns out that other activation functions behave much better in deep neural networks, in particular the ReLU activation function, mostly because it does not saturate for positive values (and also because it is quite fast to compute).

Unfortunately, the ReLU activation function is not perfect. It suffers from a problem known as the *dying ReLUs*: during training, some neurons effectively die, meaning they stop outputting anything other than 0. In some cases, you may find that half of your network's neurons are dead, especially if you used a large learning rate. During training, if a neuron's weights get updated such that the weighted sum of the neuron's inputs is negative, it will start outputting 0. When this happen, the neuron is unlikely to come back to life since the gradient of the ReLU function is 0 when its input is negative.

To solve this problem, you may want to use a variant of the ReLU function, such as the *leaky ReLU*. This function is defined as $\text{LeakyReLU}_\alpha(z) = \max(\alpha z, z)$ (see Figure 11-2). The hyperparameter α defines how much the function "leaks": it is the slope of the function for $z < 0$, and is typically set to 0.01. This small slope ensures that leaky ReLUs never die; they can go into a long coma, but they have a chance to eventually wake up. A recent paper[5] compared several variants of the ReLU activation function and one of its conclusions was that the leaky variants always outperformed the strict ReLU activation function. In

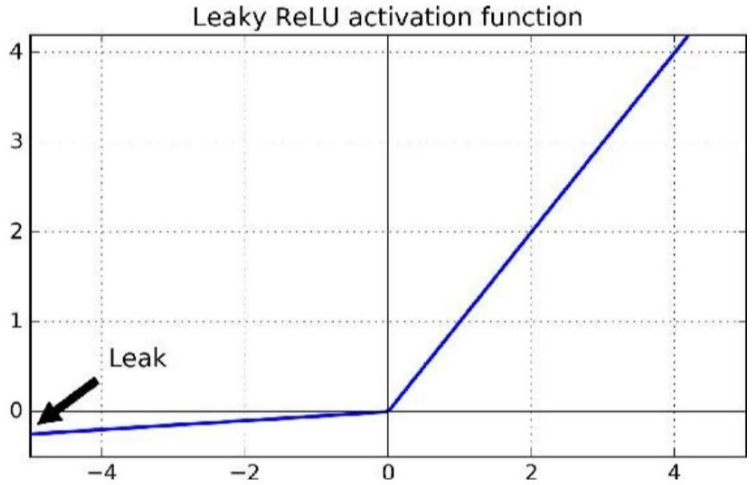


Figure 11-2. Leaky ReLU

Transfer learning

Go to page 357

retrained Layers

It is generally not a good idea to train a very large DNN from scratch: instead, you should always try to find an existing neural network that accomplishes a similar task to the one you are trying to tackle, then just reuse the lower layers of this network: this is called *transfer learning*. It will not only speed up training considerably, but will also require much less training data.

For example, suppose that you have access to a DNN that was trained to classify pictures into 100 different categories, including animals, plants, vehicles, and everyday objects. You now want to train a DNN to classify specific types of vehicles. These tasks are very similar, so you should try to reuse parts of the first network (see Figure 11-4).

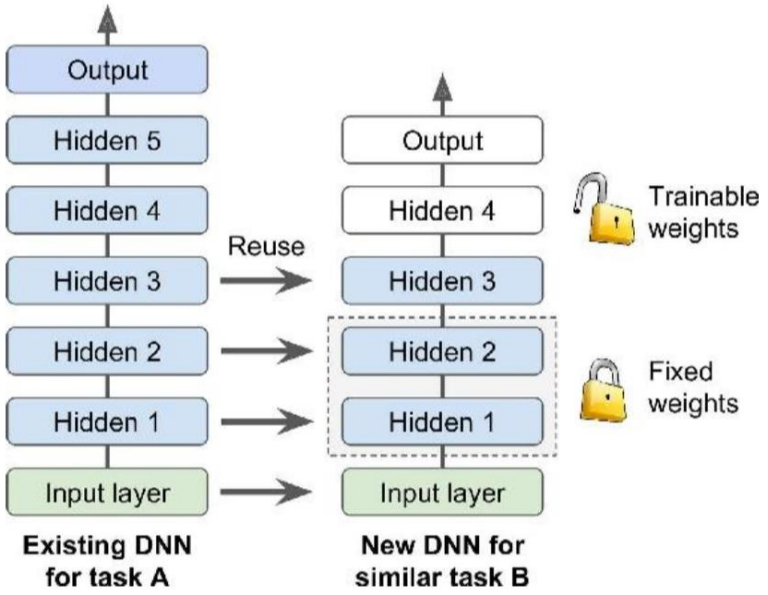


Figure 11-4. Reusing pretrained layers

Dropout

The most popular regularization technique for deep neural networks is arguably *dropout*. It was proposed[21] by G. E. Hinton in 2012 and further detailed in a paper[22] by Nitish Srivastava et al., and it has proven to be highly successful: even the state-of-the-art neural networks got a 1–2% accuracy boost simply by adding dropout. This may not sound like a lot, but when a model already has 95% accuracy, getting a 2% accuracy boost means dropping the error rate by almost 40% (going from 5% error to roughly 3%).

It is a fairly simple algorithm: at every training step, every neuron (including the input neurons but excluding the output neurons) has a probability *p* of being temporarily "dropped out," meaning it will be entirely ignored during this training step, but it may be active during the next step (see Figure 11-9). The hyperparameter *p* is called the *dropout rate*, and it is typically set to 50%. After training, neurons don't get dropped anymore. And that's all (except for a technical detail we will discuss momentarily).

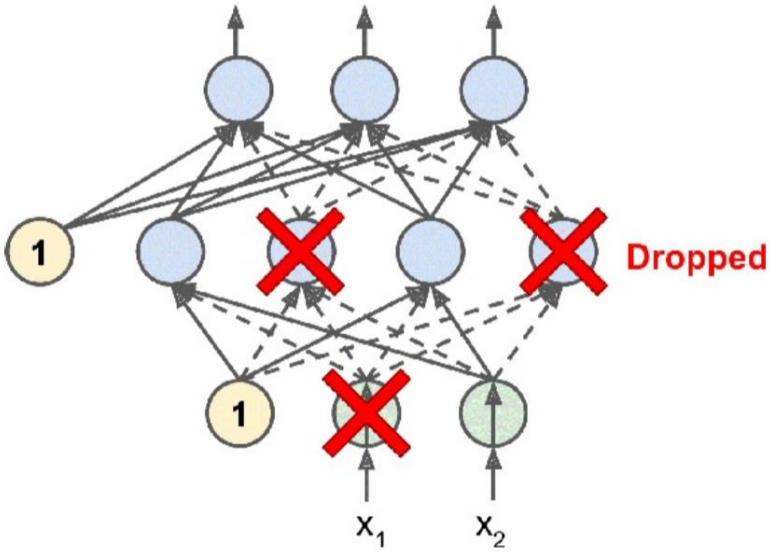


Figure 11-9. Dropout regularization

Data Augmentation

One last regularization technique, data augmentation, consists of generating new training instances from existing ones, artificially boosting the size of the training set. This will reduce overfitting, making this a regularization technique. The trick is to generate realistic training instances; ideally, a human should not be able to tell which instances were generated and which ones were not. Moreover, simply adding white noise will not help; the modifications you apply should be learnable (white noise is not).

For example, if your model is meant to classify pictures of mushrooms, you can slightly shift, rotate, and resize every picture in the training set by various amounts and add the resulting pictures to the training set (see Figure 11-10). This forces the model to be more tolerant to the position, orientation, and size of the mushrooms in the picture. If you want the model to be more tolerant to lighting conditions, you can similarly generate many images with various contrasts. Assuming the mushrooms are symmetrical, you can also flip the pictures horizontally. By combining these transformations you can greatly increase the size of your training set.

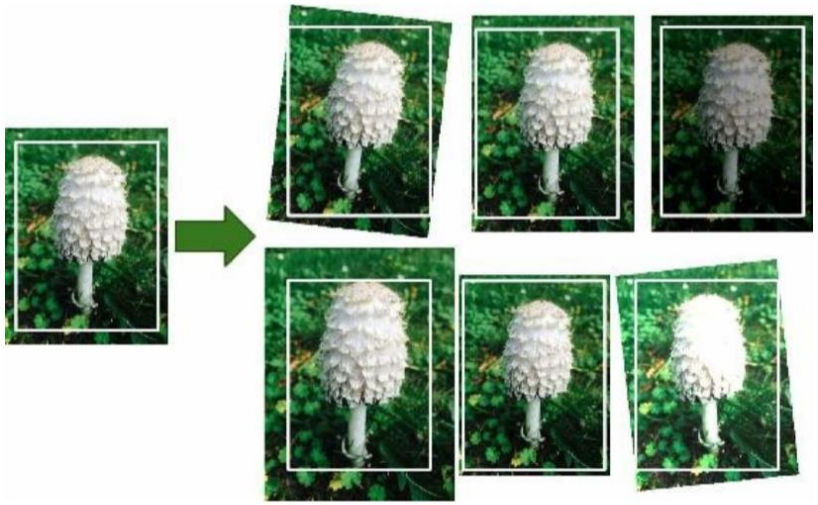


Figure 11-10. Generating new training instances from existing ones

Convolutional neural networks (CNNs) emerged from the study of the brain's visual cortex, and they have been used in image recognition since the 1980s. In the last few years, thanks to the increase in computational power, the amount of available training data, and the tricks presented in Chapter 11 for training deep nets, CNNs have managed to achieve superhuman performance on some complex visual tasks. They power image search services, self-driving cars, automatic video classification systems, and more. Moreover, CNNs are not restricted to visual perception: they are also successful at other tasks, such as *voice recognition* or *natural language processing* (NLP); however, we will focus on visual applications for now.

The Architecture of the Visual Cortex

David H. Hubel and Torsten Wiesel performed a series of experiments on cats in 1958[1] and 1959[2] (and a few years later on monkeys[3]), giving crucial insights on the structure of the visual cortex (the authors received the Nobel Prize in Physiology or Medicine in 1981 for their work). In particular, 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 (see Figure 13-1, 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. Moreover, the authors showed that some neurons react only to images of horizontal lines, while others react only to lines with different orientations (two neurons may have the same receptive field but react to different line orientations). 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 Figure 13-1, notice that each neuron is connected only to a few neurons from the previous layer). This powerful architecture is able to detect all sorts of complex patterns in any area of the visual field.

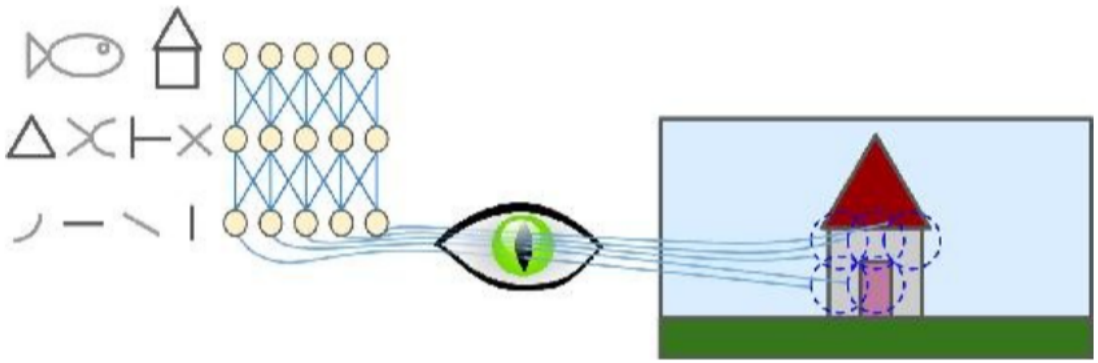


Figure 13-1. Local receptive fields in the visual cortex

> These studies of the visual cortex inspired the neocognitron, introduced in 1980,[4] which gradually evolved into what we now call *convolutional neural networks*. An important milestone was a 1998 paper[5] by Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner, which introduced the famous *LeNet-5* architecture, widely used to recognize handwritten check numbers. This architecture has some building blocks that you already know, such as fully connected layers and sigmoid activation functions, but it also introduces two new building blocks: *convolutional layers* and *pooling layers*. Let's look at them now.

Convolutional Layer

The most important building block of a CNN is the *convolutional layer*:[6] neurons in the first convolutional layer are not connected to every single pixel in the input image (like they were in previous chapters), but only to pixels in their receptive fields (see Figure 13-2). 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 low-level features in the first hidden layer, then assemble them into 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.

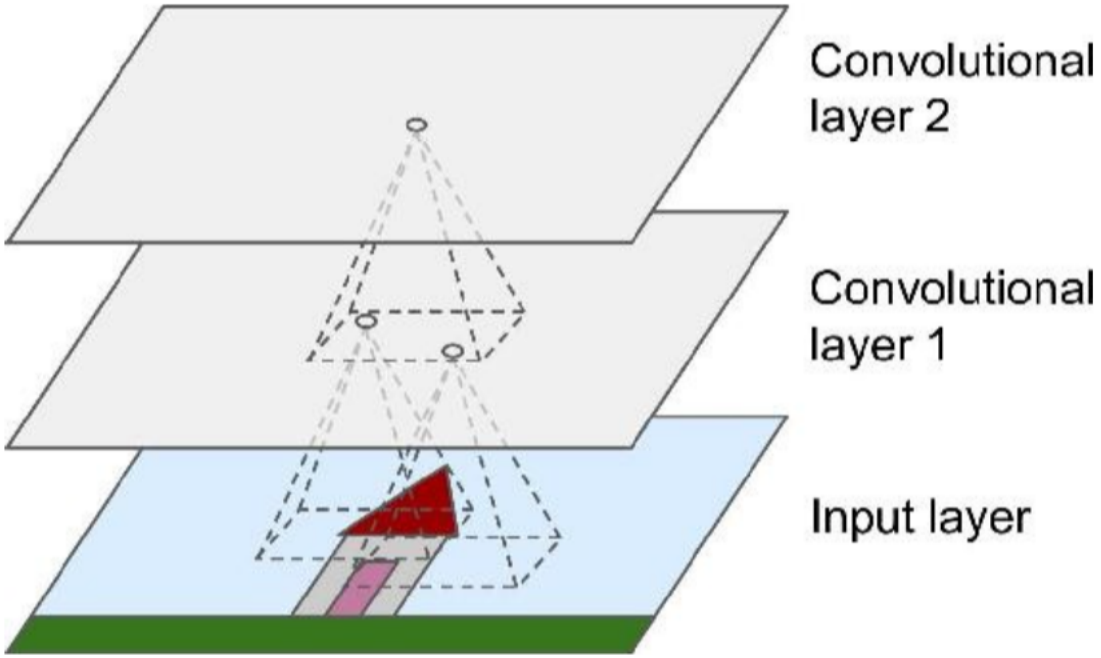


Figure 13-2. CNN layers with rectangular local receptive fields

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 (see Figure 13-3). In order for a layer to have the same height and width as the previous layer, it is common to add zeros around the inputs, as shown in the diagram. This is called *zero padding*.

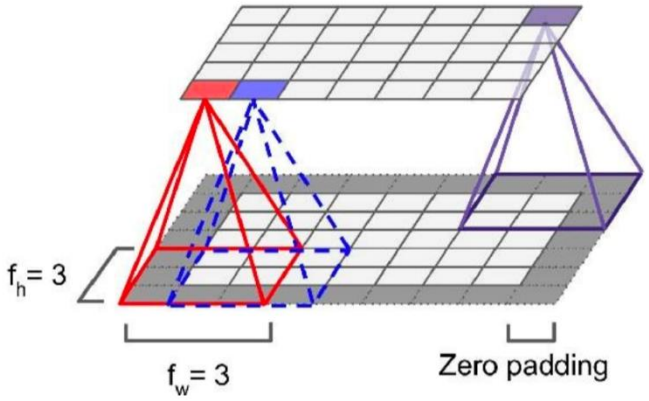


Figure 13-3. Connections between layers and zero padding

It is also possible to connect a large input layer to a much smaller layer by spacing out the receptive fields, as shown in Figure 13-4. The distance between two consecutive receptive fields is called the *stride*. In the diagram, a 5 × 7 input layer (plus zero padding) is connected to a 3 × 4 layer, using 3 × 3 receptive fields and a stride of 2 (in this example the stride is the same in both directions, but it does not have to be so). 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 $i \times s_h$ to $i \times s_h + f_h - 1$, columns $j \times s_w + f_w - 1$, where s_h and s_w are the vertical and horizontal strides.

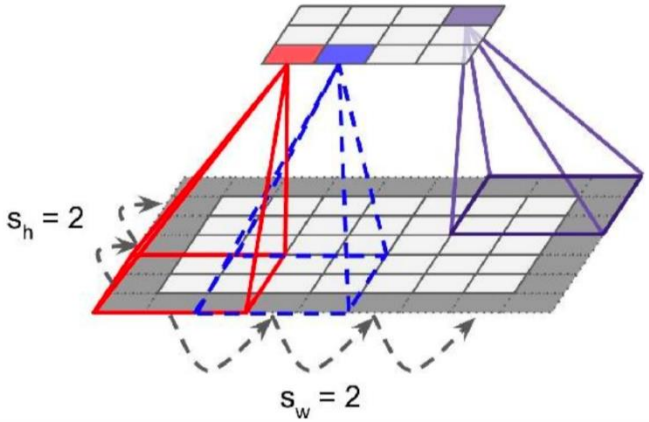


Figure 13-4. Reducing dimensionality using a stride

https://hannibunny.github.io/mlbook/neuralnetworks/convolutionDemos.html

Filters

A neuron's weights can be represented as a small image the size of the receptive field. For example, Figure 13-5 shows two possible sets of weights, called *filters* (or *convolution kernels*). The first one is represented as a black square with a vertical white line in the middle (it is a 7 × 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 (since all inputs will get multiplied by 0, except for the ones located in the central vertical line). The second filter is a black square with a horizontal white line in the middle. Once again, neurons using these weights will ignore everything in their receptive field except for the central horizontal line.

Now if all neurons in a layer use the same vertical line filter (and the same bias term), and you feed the network the input image shown in Figure 13-5 (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 you get if all neurons use the horizontal line filter; notice that the horizontal white lines get enhanced while the rest is blurred out. Thus, a layer full of neurons using the same filter gives you a *feature map*, which highlights the areas in an image that are most similar to the filter. During training, a CNN finds the most useful filters for its task, and it learns to combine them into more complex patterns (e.g., a cross is an area in an image where both the vertical filter and the horizontal filter are active).

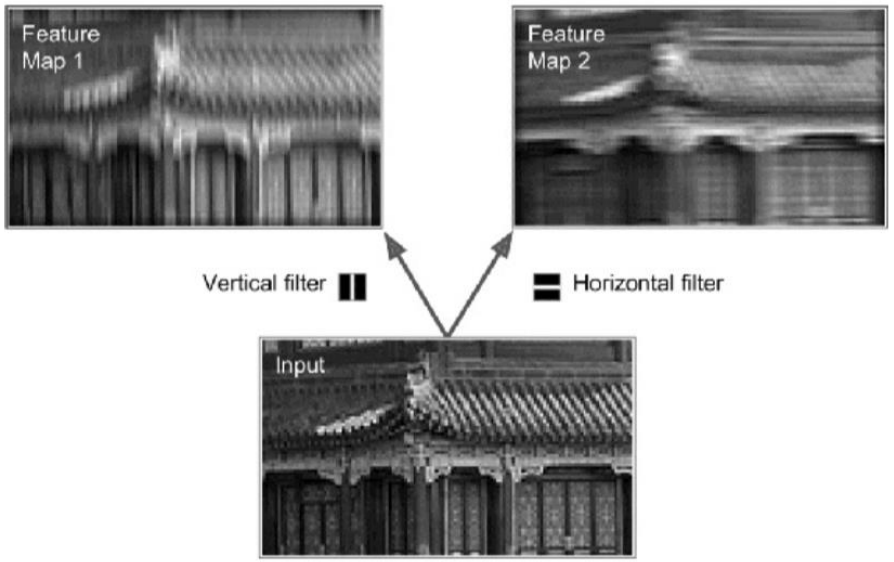


Figure 13-5. Applying two different filters to get two feature maps

Stacking Multiple Feature Maps

Up to now, for simplicity, we have represented each convolutional layer as a thin 2D layer, but in reality it is composed of several feature maps of equal sizes, so it is more accurately represented in 3D (see Figure 13-6). Within one feature map, all neurons share the same parameters (weights and bias term), but different feature maps may have different parameters. A neuron's receptive field is the same as described earlier, but it extends across all the previous layers' feature maps. In short, a convolutional layer simultaneously applies multiple filters to its inputs, making it capable of detecting multiple features anywhere in its inputs.

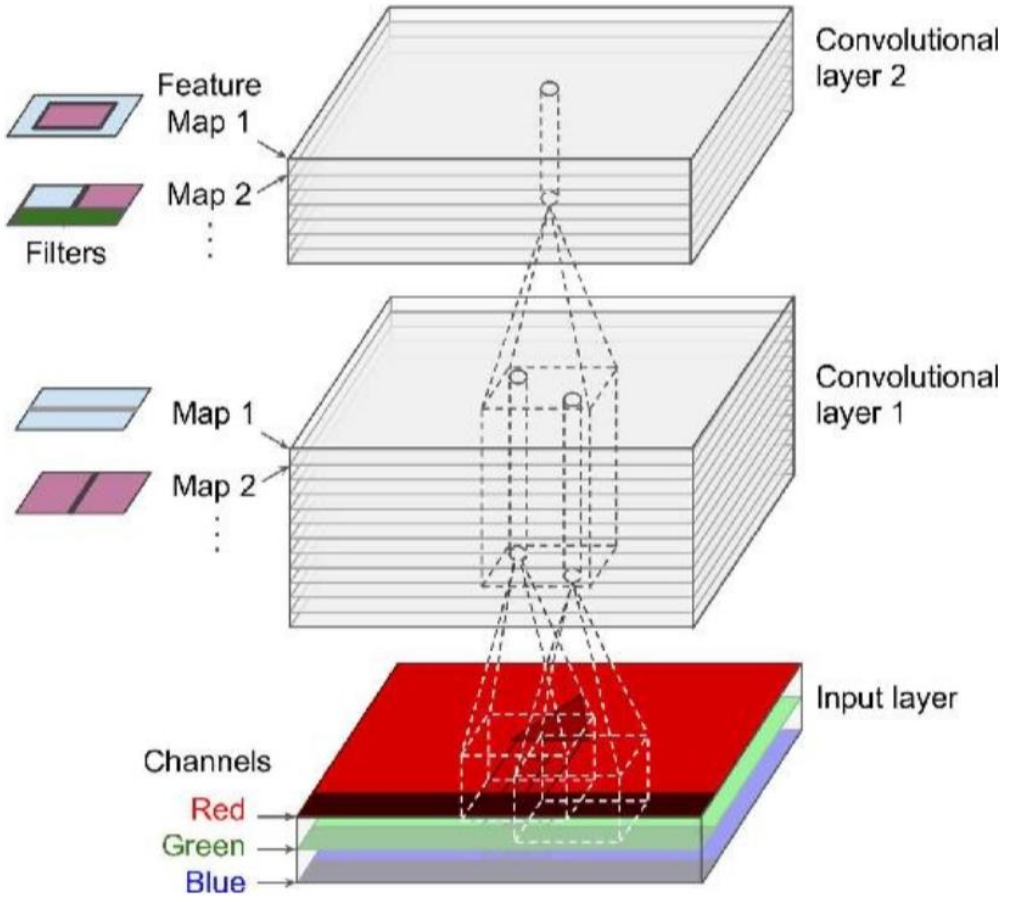


Figure 13-6. Convolution layers with multiple feature maps, and images with three channels

Pooling Layer

Once you understand how convolutional layers work, the pooling layers are quite easy to grasp. Their goal is to *subsample* (i.e., shrink) the input image in order to reduce the computational load, the memory usage, and the number of parameters (thereby limiting the risk of overfitting). Reducing the input image size also makes the neural network tolerate a little bit of image shift (*location invariance*).

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. You 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. Figure 13-8 shows a *max pooling layer*, which is the most common type of pooling layer. In this example, we use a 2 × 2 *pooling kernel*, a stride of 2, and no padding. Note that only the max input value in each kernel makes it to the next layer. The other inputs are dropped.

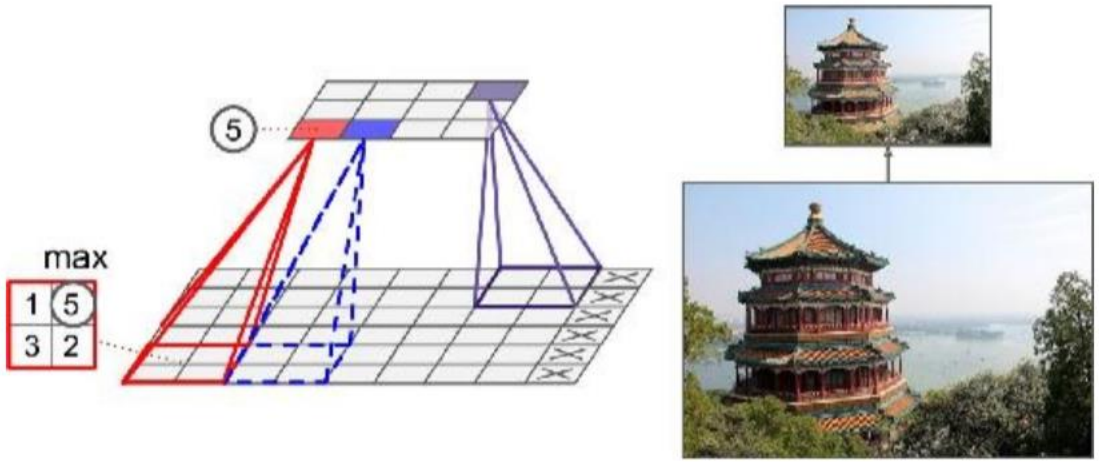


Figure 13-8. Max pooling layer (2 × 2 pooling kernel, stride 2, no padding)

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 Figure 13-9). 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).

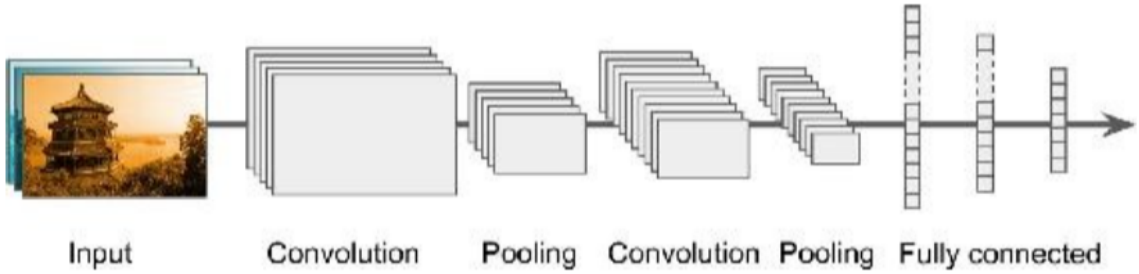


Figure 13-9. Typical CNN architecture

LeNet-5

The LeNet-5 architecture is perhaps the most widely known CNN architecture. As mentioned earlier, it was created by Yann LeCun in 1998 and widely used for handwritten digit recognition (MNIST). It is composed of the layers shown in Table 13-1.

Table 13-1. LeNet-5 architecture

Layer	Type	Maps	Size	Kernel size	Stride	Activation
Out	Fully Connected	–	10	–	–	RBF
F6	Fully Connected	–	84	–	–	tanh
C5	Convolution	120	1 × 1	5 × 5	1	tanh
S4	Avg Pooling	16	5 × 5	2 × 2	2	tanh
C3	Convolution	16	10 × 10	5 × 5	1	tanh
S2	Avg Pooling	6	14 × 14	2 × 2	2	tanh
C1	Convolution	6	28 × 28	5 × 5	1	tanh
In	Input	1	32 × 32	–	–	–

AlexNet

The *AlexNet* CNN architecture[9] won the 2012 ImageNet ILSVRC challenge by a large margin: it achieved 17% top-5 error rate while the second best achieved only 26%! It was developed by Alex Krizhevsky (hence the name), Ilya Sutskever, and Geoffrey Hinton. It is quite similar to LeNet-5, only much larger and deeper, and it was the first to stack convolutional layers directly on top of each other, instead of stacking a pooling layer on top of each convolutional layer. Table 13-2 presents this architecture.

Table 13-2. AlexNet architecture

Layer	Type	Maps	Size	Kernel size	Stride	Padding	Activation
Out	Fully Connected	–	1,000	–	–	–	Softmax
F9	Fully Connected	–	4,096	–	–	–	ReLU
F8	Fully Connected	–	4,096	–	–	–	ReLU
C7	Convolution	256	13 × 13	3 × 3	1	SAME	ReLU
C6	Convolution	384	13 × 13	3 × 3	1	SAME	ReLU
C5	Convolution	384	13 × 13	3 × 3	1	SAME	ReLU
S4	Max Pooling	256	13 × 13	3 × 3	2	VALID	–
C3	Convolution	256	27 × 27	5 × 5	1	SAME	ReLU
S2	Max Pooling	96	27 × 27	3 × 3	2	VALID	–
C1	Convolution	96	55 × 55	11 × 11	4	SAME	ReLU
In	Input	3 (RGB)	224 × 224	–	–	–	–

GoogLeNet

The GoogLeNet architecture was developed by Christian Szegedy et al. from Google Research,[10] and it won the ILSVRC 2014 challenge by pushing the top-5 error rate below 7%. This great performance came in large part from the fact that the network was much deeper than previous CNNs (see Figure 13-11). This was made possible by sub-networks called *inception modules*,[11] which allow GoogLeNet to use parameters much more efficiently than previous architectures: GoogLeNet actually has 10 times fewer parameters than AlexNet (roughly 6 million instead of 60 million).

Figure 13-10 shows the architecture of an inception module. The notation "3 × 3 + 2(S)" means that the layer uses a 3 × 3 kernel, stride 2, and SAME padding. The input signal is first copied and fed to four different layers. All convolutional layers use the ReLU activation function. Note that the second set of convolutional layers uses different kernel sizes (1 × 1, 3 × 3, and 5 × 5), allowing them to capture patterns at different scales. Also note that every single layer uses a stride of 1 and SAME padding (even the max pooling layer), so their outputs all have the same height and width as their inputs. This makes it possible to concatenate all the outputs along the depth dimension in the final *depth concat layer* (i.e., stack the feature maps from all four top convolutional layers). This concatenation layer can be implemented in TensorFlow using the `tf.concat()` operation, with `axis=3` (axis 3 is the depth).

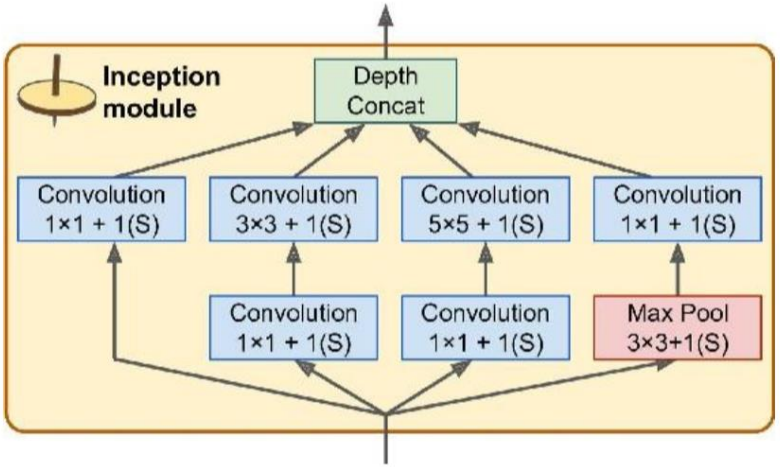


Figure 13-10. Inception module

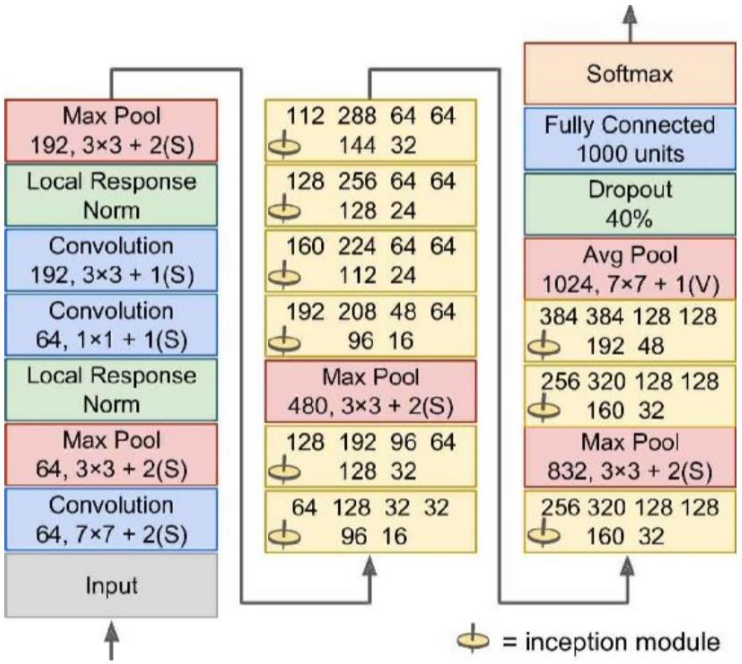


Figure 13-11. GoogLeNet architecture

ResNet

Last but not least, the winner of the ILSVRC 2015 challenge was the *Residual Network* (or *ResNet*), developed by Kaiming He et al.,[12] which delivered an astounding top-5 error rate under 3.6%, using an extremely deep CNN composed of 152 layers. The key to being able to train such a deep network is to use *skip connections* (also called *shortcut connections*): the signal feeding into a layer is also added to the output of a layer located a bit higher up the stack. Let's see why this is useful.

When training a neural network, the goal is to make it model a target function $h(\mathbf{x})$. If you add the input $\mathbf{x}$ to the output of the network (i.e., you add a skip connection), then the network will be forced to model $f(\mathbf{x}) = h(\mathbf{x}) - \mathbf{x}$ rather than $h(\mathbf{x})$. This is called *residual learning* (see Figure 13-12).

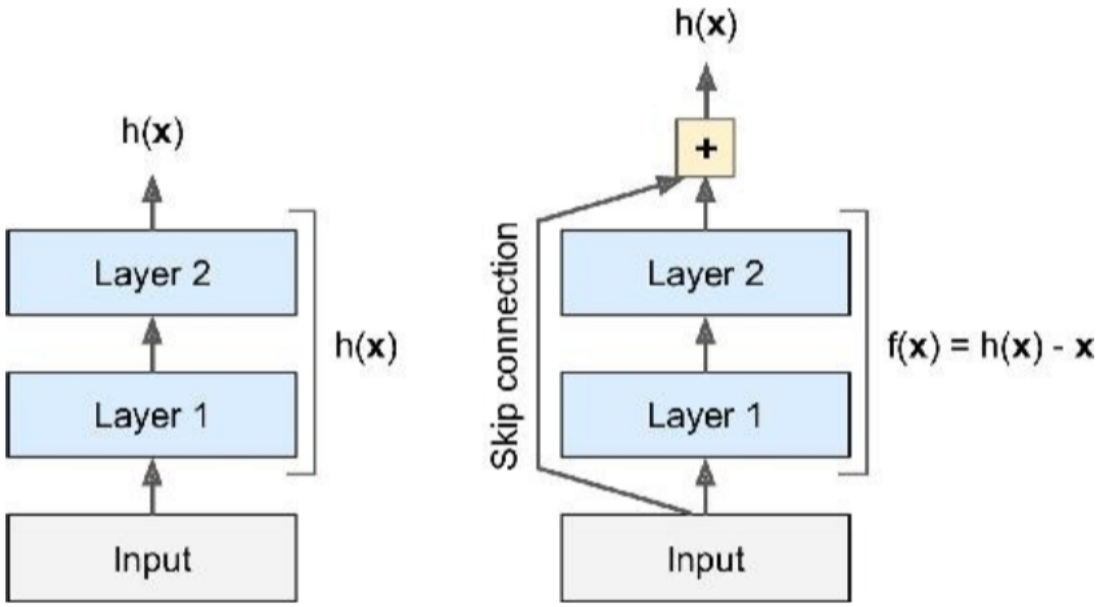


Figure 13-12. Residual learning

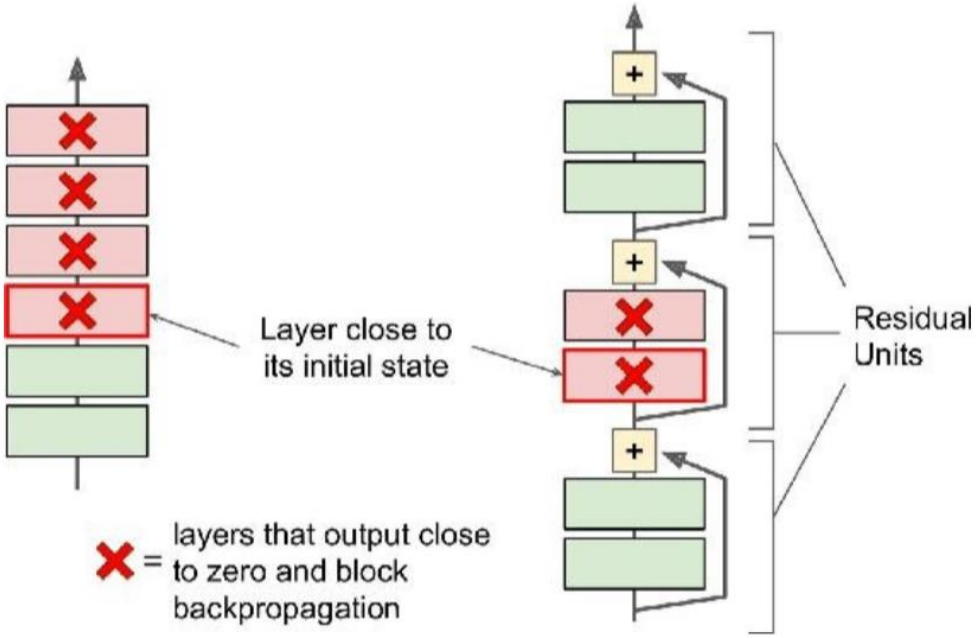
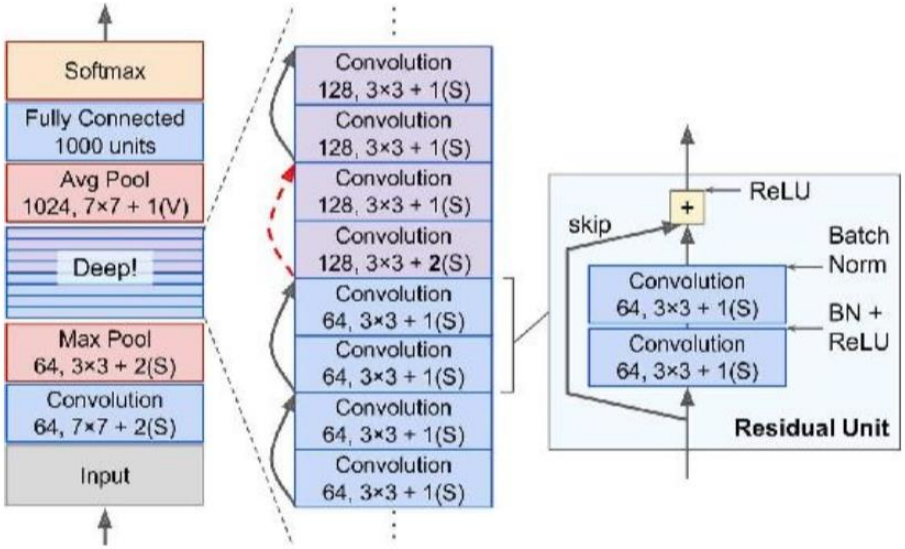


Figure 13-13. Regular deep neural network (left) and deep residual network (right)



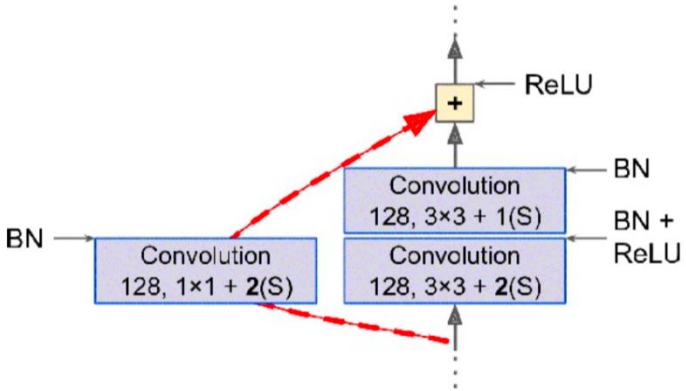
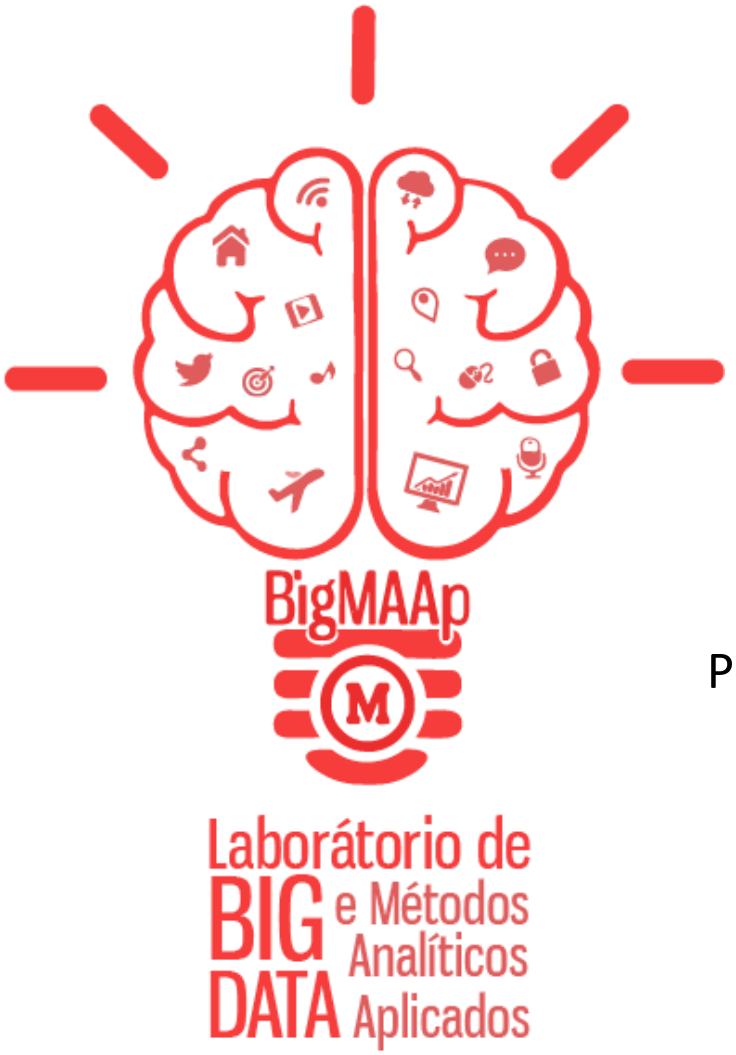


Figure 13-14. ResNet architecture



Prof. Dr. Leandro Augusto da Silva

leandroaugusto.silva@mackenzie.br

Faculdade de Computação e Informática

Programa de Pós-Graduação em Engenharia Elétrica e Computação