Learning Deep Architectures (part III)

Enrique Romero

Advanced Topics in Artificial Intelligence
Master in Artificial Intelligence
Soft Computing Group
Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya, Barcelona, Spain

2017-2018
2012-2013 (first course)

Outline

Outline

A Bit of History

Convolutional Neural Networks (CNNs) are very old models:

- Neocognitron [Fukushima, 1980] incorporated most of the elements in modern CNNs, but wit a different approach
- Time-Delay Neural Networks [Lang and Hinton, 1988, Lang et al., 1990] are one-dimensional versions of CNNs applied to time series, trained with back-propagation
- Convolutional (Neural) Networks are described for the first time in [LeCun et al., 1989]

A Bit of History

CNNs were some of the first neural networs to be used in commercial applications:

- AT&T developed a CNN for reading checks [LeCun et al., 1998], so that by the end of the 1990s this system was reading over 10% of all checks in the USA
- Microsoft deployed several OCR and handwritting recognition systems based on CNNs [Simard et al., 2003]
- More information: [LeCun et al., 2010]

More recently, CNNs have attracted much attention because they have been used by the winner models of several important contests (the first one was the 2012 ImageNet object recognition challenge [Krizhevsky et al., 2012])

Outline

Main Ideas and Motivation

A (simple) definition: **CNNs are Neural Networks that compute a convolution instead of a matrix multiplication in at least one of their layers**

The convolution operation implies that we have:

- **Sparse interactions** (sparse connectivity)
- **Weight sharing**

with the aim of obtaining **translation invariance** (be able to find a pattern in any place of the input)

Typically, the convolution operation is combined with other operations, such as **non-linear transformation** and **pooling**

Outline

Definition of Convolution

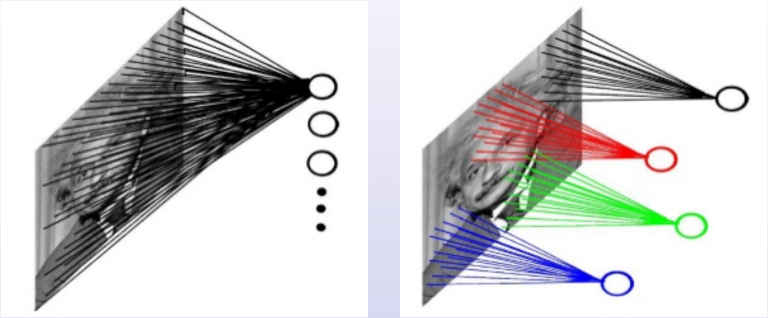


Figure 1 : Difference between classical MLP and CNN hidden units

With a convolution, the **same weights** are used to compute output values in **different and small parts** of the input

Definition of Convolution

Mathematical definition of the convolution of two functions F, W:

$$C(t) = (F{*}W)(t) = \int_{-\infty}^{\infty} F(t{-}x)\, W(x)\, dx = \int_{-\infty}^{\infty} F(x)\, W(t{-}x)\, dx$$

Interpretation: We can think that the convolution $(F * W)(t)$ is an expectation / weighted average (by W) of F around t

Discrete convolution used in 2D CNNs (*cross-correlation*):

$$C(n, m) = (I * K)(n, m) = \sum_i \sum_j I(n + i, m + j) K(i, j)$$

where

- I is the **input**, K is the **feature detector** or **filter** or **kernel** (the weights) and C is the **feature map** (the output)
- The sum is defined over all valid values

Viewing the Convolution in a Classical FNN

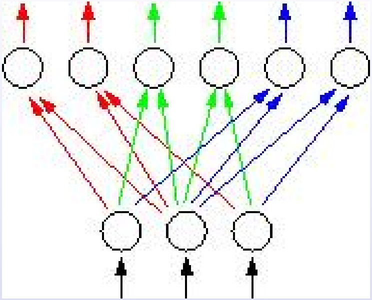


Figure 2 : Viewing CNNs as Classical FNNs

Note that we have:

- **Sparse interactions** (sparse connectivity)
- **Weight sharing**

with the aim of obtaining **translation invariance**

In addition, it has **much less parameters** than a classical fully connected network

Viewing the Convolution in a Classical FNN

A convolutional operation is similar to a **local receptive field** with shared weights

The previous figure shows us that **discrete convolution can be seen as a multiplication by a matrix with restrictions**:

- Equal weights in different units
- Zeros for several weights

Although it is not used in practice (it would be very inefficient) it allows to understand it as a classical Feed-forward Neural Network (FNN) and **apply standard techniques (back-propagation, for example) in a natural way**

Obviously, it needs to **reshape** the input data (e.g., an image)

Outline

Working with Channels

In real data sets, there are additional dimensions where we do not want to apply the convolution operation:

- 1D: Multi-dimensional time series (for example, animations of "skeletons", where the pose is described by the angles of each joint in the skeleton)
- 2D: Color images (each pixel is a rgb vector)
- 3D: Color volumetric data (for example, medical rgb CT scans)

These additional dimensions are usually called **channels** (also referred to as the **depth** of the input)

How does the convolution operate with channels?

Working with Channels

For example, in 2D color images (3 channels), we have:

- The input is a $N \times M \times 3$ tensor
- Every filter K_f is a $A \times B \times 3$ tensor
- The convolution of the input image I and the filter K_f is a $N \times M \times 1$ tensor (a matrix), where each component is:

$$C_{K_f}(n,m) = \sum_{i=1}^{A} \sum_{j=1}^{B} \sum_{k=1}^{3} I(n+i, m+j, k) K_f(i,j,k)$$

- Every filter "outputs an image of one channel": If we have F filters, the output of the convolutional layer is a $N \times M \times F$ tensor, where the third component has the convolutions of the image with every filter:

$$C(n,m,f) = C_{K_f}(n,m)$$

Working with Channels

Therefore, **the outputs of a convolutional layer, in turn, are also reshaped in a structure with channels**, so that it can be used as the input of a new convolutional layer:

- The number of dimensions of the original image where convolution applies is constant between additional layers (N and M)
- The number of output channels of a convolutional layers is the number of filters (F)

This **allows to construct deep CNNs in a natural way**

Outline

Working with Bias

We can add some bias terms to the convolution

How do they operate?

Typically, we will have one bias per channel in the output (one bias per filter), shared across all locations in the convolution

Outline

Hyperparameters of Convolutions

The first hyperparameter is, obviously, **the size of the filter $A \times B$**

With the previous definition of convolution, the size of the output of the convolution of a $N \times M$ image with an $A \times B$ filter is exactly $(N - A + 1) \times (M - B + 1)$, since only valid positions can be used to compute the convolution

We may want:

- On the one hand, to have the possibility to obtain outputs with the same size than the original one by adding zeros (**Zero-padding**)
- On the other hand, reduce the computational cost by removing (or not computing) some of the outputs of the convolution (**Stride**)

Hyperparameters of Convolutions: Zero-padding

The most common settings for Zero-padding are

- When Zero-padding = *valid*, the convolution is only allowed to visit positions where the entire filter is entirely within the image (and therefore reducing the size of the output)
- When Zero-padding = *same*, zeros are added at the borders so that the size of the output is equal to the input

0	0	0	0	0	0
0	35	19	25	6	0
0	13	22	16	53	0
0	4	3	7	10	0
0	9	8	1	3	0
0	0	0	0	0	0

Figure 3 : Convolution with Zero-padding

Hyperparameters of Convolutions: Stride

The easiest and most common way to reduce the outputs of a convolution is to sample only every s pixels in each direction

It is a **downsampled convolution**:

$$C(n, m) = \sum_i \sum_j I(n + i \times s, m + j \times s) K(i, j)$$

The parameter s is the *stride* of the downsampled convolution

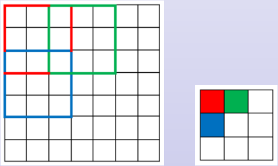


Figure 4 : Convolution with Stride = 2

Some information is lost, but the computations are cheaper

Outline

Additional Operations and Layers in CNNs

CNNs may have other types of operations and layers:

- Handle inputs of variable size: applying each filter a different number of times depending on the size of the input
- Non-linearities: apply a non-linear transformation to the output of the convolution (sigmoidal, ReLU,...)
- Pooling layers (see next slide)
- Fully connected layers: standard layers (typically on top of the convolutional layers)

Pooling layers **replace the output of the network at a certain rectangle with a summary statistic of its contents**

The most common pooling functions are:

- Max-pooling: the output is the maximum value
- Average-pooling: the output is the mean value

Hyperparameters of the pooling operation:

- The size of the rectangle (typically 2×2)
- The stride (typically 2)

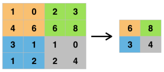


Figure 5 : Max-pooling of size 2×2 and stride $= 2$

Additional Operations and Layers in CNNs: Pooling layers

Pooling layers are used to reduce the spatial size of the representation and the number of parameters, thus reducing the amount of computation in the network

For example, with a pooling of size of 2×2 and stride 2 the number of outputs is reduced a 75%

Additionally, it makes the representation approximately invariant to small translations of the input (i.e., if we translate the input by a small amount, the values of most of the outputs do not change)

Typical Architectures of CNNs

In summary, in a CNN we can find:

- Convolutional layers
- Non-linearity layers (transformations of the convolution)
- Pooling layers
- Fully connected layers

Typically:

- A pooling layer is inserted after several Convolutional + Non-linearity layers (**deep feature learning/extraction**)
- Fully connected layers are the output layers of the network (**discrimination**)

Typical Architectures of CNNs

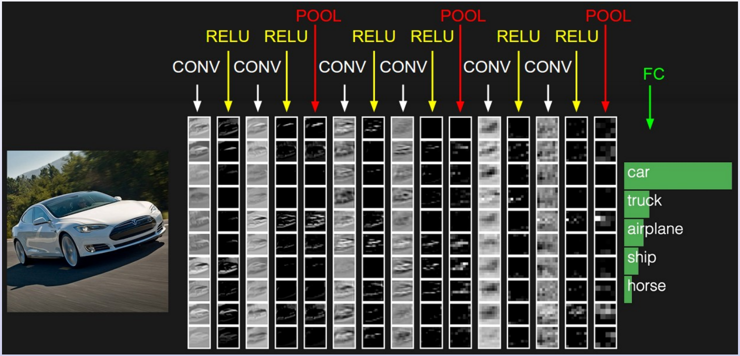


Figure 6 : Typical architecture of a CNN (layers)

Typical Architectures of CNNs

What about the sizes? A common practice is to divide the size of the inputs by a factor and multiply the depth by another factor, forming a pyramid

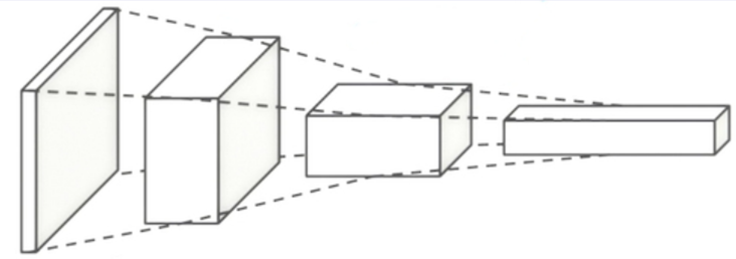


Figure 7 : Typical architecture of a CNN (sizes)

Several Famous Architectures of CNNs

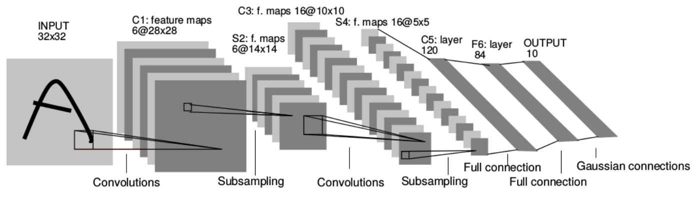


Figure 8 : LeNet-5 architecture [LeCun et al., 1998]

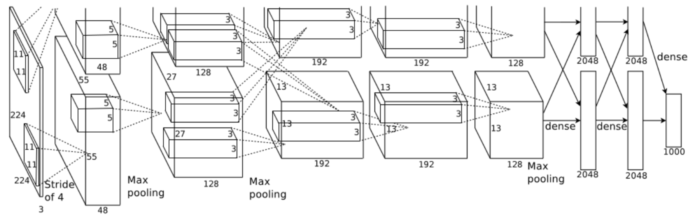


Figure 9 : AlexNet architecture [Krizhevsky et al., 2012]

Several Famous Architectures of CNNs

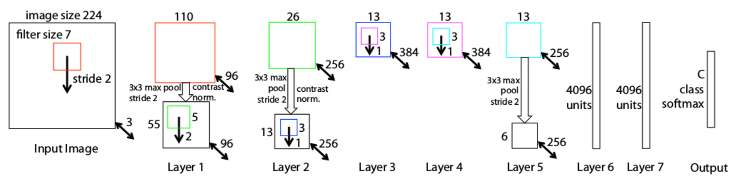


Figure 10 : ZFNet architecture [Zeiler and Fergus, 2014]

More famous CNNs:
`http://cs231n.github.io/convolutional-networks`

Outline

Training CNNs with Back-Propagation

Since CNNs can be viewed as classical FNNs (see above), back-propagation can be easily applied to CNNs:

- For shared weights in convolutional layers, the derivative is the sum of the back-propagated derivatives
- For non-linearities, the derivative back-propagates as usual
- For pooling layers, the derivative is back-propagated according to the type of pooling and forward propagations (max-pooling only propagates the derivative of the maximum element, etc)
- For fully connected layers, the derivative back-propagates as usual

Outline

What Makes CNN Work?

Basically, the joint interaction of a number of ingredients:

- Good initialization of the weights [Glorot and Bengio, 2010]
- Non-saturated activation functions, such as Rectified Linear Units (ReLUs) $f(x) = max(0, x)$ (or any of its variants) [Nair and Hinton, 2010, Glorot et al., 2011]
- Adaptive learning rates (RMSProp, Adagrad,...) [Tieleman and Hinton, 2012, Duchi et al., 2011]
- Strong regularization techniques, such as dropout [Srivastava et al., 2014] or other tricks, such as batch normalization [Ioffe and Szegedy, 2015] or local contrast normalization
- High performance resources (GPUs, supercomputers)
- A large set of labeled examples
- Data where the convolution operation make sense
- etc

Outline

Do It Yourself Deep Learning

Recently, the idea that anyone can build deep learning models is thoroughly extended: It is the "Do It Yourself Deep Learning"

This is basically due to three facts:

- Personal computers can be equipped with specific high performance hardware/software for deep networks (GPUs/Cuda)
- It is relatively easy to obtain large quantities of data (internet)
- There exist software tools that allow to construct very complex models very easily

Outline

Several Software Tools that Help

Remember that, according to Geoffrey Hinton, the data more suitable for deep architectures should have these properties:

- High-dimensional data (hundreds, thousands, or even more)
- Noise is not the main problem in the data
- The data is structured, but this structure is difficult to represent in a simple model
- We have a VERY large amount of available data (to capture the many factors of variation in real data)

With this kind of data, **the main problem is to imagine how can we represent this complex structure so that it can be learned**

Several Software Tools that Help

Suppose that we have already imagined how to represent the complex structure of the data and we have an idea of the architecture we want to construct...

- How long will take to implement it?
- How long will take to check the implementation is correct?
- How long will take to implement and check the modifications to the original idea?
- How long will take the experiments to adjust the architecture parameters and other hyperparameters?

Several Software Tools that Help

Currently, we have a number of software tools that make your life easier, since

- They offer a number of **efficiently implemented** types of units and layers (classical MLP, convolutional, recurrent) with many variants
- They offer a number of already implemented **activation functions**
- They offer a number of already implemented **optimization schemes**
- They offer **automatic/symbolic differentiation**, so that you do not even need to implement the derivatives

Several Software Tools that Help

With this kind of software, you “only” have to

- Define your architecture:
  - The weights of every layer
  - The layers and the computations in every layer (the units)
  - The forward propagation of the activation along the network (the connections across layers)
- Define your loss function
- Select an optimization scheme

and the system will **efficiently** do the rest:

- Compute the loss function on your data
- Compute the derivatives of the loss function wrt the weights
- Apply the optimization scheme to change the weights

Note that you only have to **DEFINE**, not implement

Several Software Tools that Help

Several examples of these software tools are:

- TensorFlow (Google)
- Theano (University of Montreal)
- Torch (R. Collobert, Facebook AI Research)
- CNTK (Microsoft)
- MXNet (Distributed ML Community)
- Keras (F. Chollet, Google engineer)

More information at `https://en.wikipedia.org/wiki/Comparison_of_deep_learning_software`

Although they may have a steep learning curve, the benefits may be large in the end

Outline

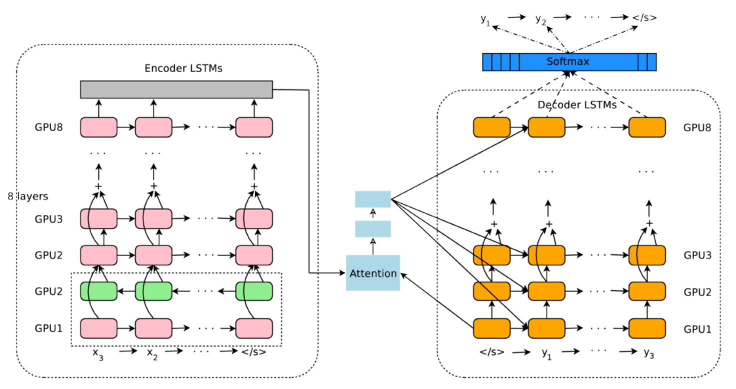
Do It Yourself Deep Learning?

An example: Google Neural Machine Translation [Wu et al., 2016]

Let's imagine how can we represent the complex structure of the data (in authors' words): "Our model follows the common sequence-to-sequence learning framework with attention. It has three components: an encoder network, a decoder network, and an attention network. The encoder transforms a source sentence into a list of vectors, one vector per input symbol. Given this list of vectors, the decoder produces one symbol at a time, until the special end-of-sentence symbol (EOS) is produced. The encoder and decoder are connected through an attention module which allows the decoder to focus on different regions of the source sentence during the course of decoding".

Do It Yourself Deep Learning?

The architecture:



As you can see, imagine how can we represent the complex structure of the data is not easy at all...

Outline

A Critical Discussion about Deep Learning

For a critical discussion about Deep Learning, see [Marcus, 2018]

Are you interested in Deep Learning?

Are you looking for a Master Thesis Project?

That's it!

Bibliography

- Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research*, 12:2121–2159.
- Fukushima, K. (1980). Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position. *Biological Cybernetics*, 36:193–202.
- Glorot, X. and Bengio, Y. (2010). Understanding the Difficulty of Training Deep Feedforward Neural Networks. In *International Conference on Artificial Intelligence and Statistics*, pages 249–256.
- Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep Sparse Rectifier Neural Networks. In *International Conference on Artificial Intelligence and Statistics*, pages 315–323.
- Ioffe, S. and Szegedy, C. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *32th International Conference on Machine Learning*.
- Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, volume 24, pages 1106–1114. MIT Press.
- Lang, K. J. and Hinton, G. E. (1988). The Development of the Time-delay Neural Network Architecture for Speech Recognition. Technical Report CMU-CS-88-152, Carnegie-Mellon University.
- Lang, K. J., Waibel, A. H., and Hinton, G. E. (1990). A Time-delay Neural Network Architecture for Isolated Word Recognition. *Neural Networks*, 3(1):23–43.
- LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989). Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation*, 1(4):541–551.
- LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11):2278–2324.
- LeCun, Y., Kavukcuoglu, K., and Farabet, C. (2010). Convolutional Networks and Applications in Vision. In *International Symposium on Circuits and Systems*, pages 253–256.

Bibliography

- Marcus, G. (2018). Deep Learning: A Critical Appraisal. *arXiv preprint arXiv:1801.00631*.
- Nair, V. and Hinton, G. E. (2010). Rectified Linear Units Improve Restricted Boltzmann Machines. In *27th International Conference on Machine Learning*, pages 807–814.
- Simard, P. Y., Steinkraus, D., and Platt, J. C. (2003). Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis. In *International Conference on Document Analysis and Recognition*, pages 958–963.
- Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15:1929–1958.
- Tieleman, T. and Hinton, G. E. (2012). Lecture 6.5-RMSProp: Divide the Gradient by a Running Average of its Recent Magnitude. *COURSERA: Neural Networks for Machine Learning*.
- Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., et al. (2016). Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *arXiv preprint arXiv:1609.08144*.
- Zeiler, M. D. and Fergus, R. (2014). Visualizing and Understanding Convolutional Networks. In *European Conference on Computer Vision (Lecture Notes in Computer Science 8689)*, pages 818–833.