Convolutional neural networks explained

Fantine Huot

ABSTRACT

Convolutional networks, also known as ConvNets, convolutional neural networks or CNNs, are a specialized kind of neural network for processing data that has a known, grid-like topology, such time-series data or image data. They have been successful in practical applications, ranging from face recognition to self-driving cars, and could be used for processing seismic data. In this paper, I explain how to build a convolutional network.

INTRODUCTION

Convolutional networks are a specific type of feedforward networks. The theory for modern convolutional networks was introduced in the 1990s (LeCun et al., 1989). Thanks to increasing computing power and labeled data volumes, within the last 5 years, convolutional networks have had a dramatic impact on computer vision (Krizhevsky et al., 2012; He et al., 2015), speech recognition (Dahl et al., 2012; Deng et al., 2010; Seide et al., 2011; Hinton et al., 2012), and image segmentation (Sermanet et al., 2013; Farabet et al., 2013; Couprie et al., 2013; Cireşan et al., 2012). Herein, I cover the basic theory for feedforward networks, then proceed to the specificities of convolutional networks.

FEEDFORWARD NETWORKS

Deep feedforward networks, also often called feedforward neural networks, or multilayer perceptrons (MLPs), are the quintessential deep learning models. The goal of a feedforward network is to approximate some function $f^\star$. For example, for a classifier, $\boldsymbol{y} = f^\star(\boldsymbol{x})$ maps an input $\boldsymbol{x}$ to a category $\boldsymbol{y}$. A feedforward network defines a mapping f such that $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$ and learns the value of the parameters $\boldsymbol{\theta}$ that result in the best function approximation.

Representation learning

Deep feedforward networks allow us to capture non-linearities by applying linear operators to transformed inputs $\phi(\boldsymbol{x})$, where ϕ is a nonlinear transformation. The strategy of deep learning is to learn ϕ from our data. This is called representation learning.

Say we wish to estimate an output $\boldsymbol{y}$ from an input $\boldsymbol{x}$. A feedforward network defines a mapping $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta}, \boldsymbol{w}) = \phi(\boldsymbol{x}; \boldsymbol{\theta})^\top \boldsymbol{w}$. We now have parameters $\boldsymbol{\theta}$ that we use to learn ϕ from a broad class of functions, and parameters $\boldsymbol{w}$ that map from $\phi(\boldsymbol{x})$ to the desired output. This approach can be very highly generic, by learning ϕ from a very broad family of functions $\phi(\boldsymbol{x}; \boldsymbol{\theta})$; while allowing us to encode our domain knowledge by designing families $\phi(\boldsymbol{x}; \boldsymbol{\theta})$ that we expect to perform well. The advantage is that we only need to define the right general function families rather than finding precisely the right function.

Network architecture

A neural network is typically represented as a collection of units that are connected in a directed acyclic graph, describing how the functions are composed together (Figure 1). The output of each layer of units becomes the input of the next layer. For example, we might have three functions $f^{(1)}, f^{(2)}$, and $f^{(3)}$ connected in a chain, to form $f(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x})))$. In this case, $f^{(1)}$ is called the first layer of the network, $f^{(2)}$ is called the second layer, and so on. The final layer of a feedforward network is called the output layer. The layers located between the input and the output layer are called hidden units. The mapping function f corresponding to the network is the result of composing together the functions that are applied at each layer of units. The depth of a network is the number of layers (omitting the input layer).

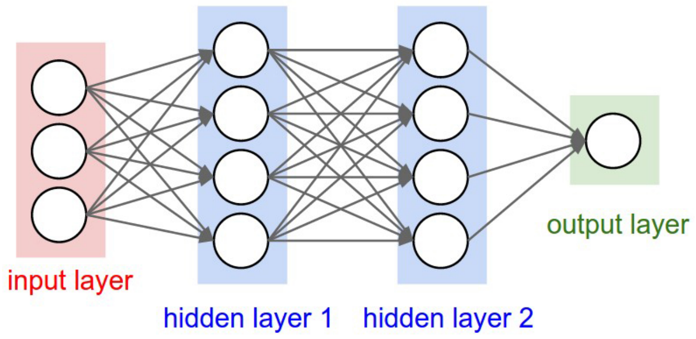

Figure 1: A 3-layer neural network with three inputs, two hidden layers of 4 units each and one output layer. (Image courtesy of Stanford CS231N). [**NR**]

Hidden units

In their most commonly used implementation, hidden units can be described as accepting a vector of inputs $\boldsymbol{x}$, and computing an affine transformation:

$$\boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b}$$

where $\boldsymbol{W}$ are weights applied to the input and $\boldsymbol{b}$ a bias term, and then applying an element-wise nonlinear function $f(\boldsymbol{z})$ called the activation function.

Activation functions

The design of hidden units is an extremely active area of research and does not yet have many definitive guiding theoretical principles. As of today, the most popular type of hidden units are rectified linear units (Figure 2) that use the activation function

$$f(\boldsymbol{z}) = \max\{0, \boldsymbol{z}\}.$$

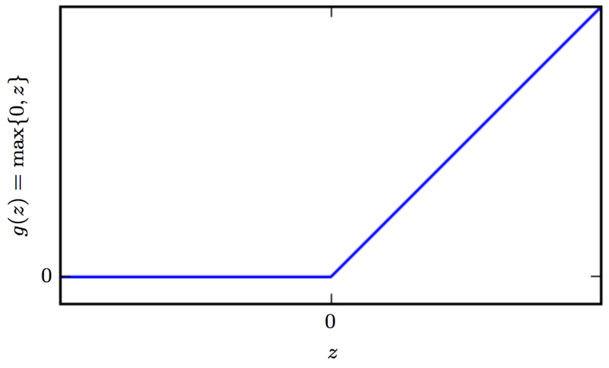

Figure 2: The rectified linear activation function. This activation function is the default activation function recommended for use with most feedforward neural networks. [**ER**]

Applying this function to the output of a linear transformation yields a nonlinear transformation. However, the function remains very close to linear, in the sense that is a piecewise linear function with two linear pieces. Because rectified linear units are nearly linear, they preserve many of the properties that make linear models easy to optimize with gradient-based methods. These are desirable properties for training neural networks, as we will see in the following sections.

Other types of activation functions that often appear in literature are the softmax function, the hyperbolic tangent function, the radial basis functions, the leaky rectified linear unit (Maas et al., 2013) and the maxout unit (Goodfellow et al., 2013). While they all have their own specificities, I will not cover them here.

Objective function

Event detection, object recognition and image segmentation are all tasks that are based on an underlying classifier.

Data loss

Let's consider a training data set $\boldsymbol{x}$ with N elements, associated with labels $\boldsymbol{y}$ corresponding to the different classes that each training data sample can belong to. The data-fitting term of the objective function measures the compatibility between the classifier's estimations and the ground truth labels. This data loss term L takes the form of an average over the data losses L_i for every training data sample i:

$$L = \frac{1}{N} \sum_i L_i.$$

We abbreviate the output of the neural network for a given input $\boldsymbol{x}_i$ as the score s:

$$s = f(\boldsymbol{x}_i; \boldsymbol{\theta}).$$

s_j, the score for the the j-th class, is the j-th element of s:

$$s_j = f(\boldsymbol{x}_i; \boldsymbol{\theta})_j.$$

For a data sample i, the score for the correct class is s_{y_i}. The two most commonly used objective functions are the softmax classifier and the multi-class support vector machine (SVM) classifier. While other formulations are possible, I will not cover them here.

Softmax classifier

The softmax classifier minimizes the cross-entropy between the estimated class probabilities and the "true" distribution (which in this interpretation is the distribution where all probability mass is on the correct class):

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_i e^{s_j}}\right).$$

SVM classifier

The multi-class support vector machine (SVM) loss uses the hinge loss, forcing the correct class for each sample to a have a score higher than the incorrect classes by some fixed margin:

$$L_i = \sum_{j \neq i} \max\{0, s_j - s_{y_i} + 1\}.$$

A detailed explanation of the SVM classifier is provided in Huot and Clapp (2016).

Regularization

To avoid overfitting and obtain good generalization results, we add regularization terms to the neural network architecture. The regularization terms can be added both at the output level or at the level of a hidden layer. When they are added at the output level, the objective function J can be expressed as the sum of the data loss and the regularization terms $\Gamma(\boldsymbol{\theta})$: $J = L + \lambda\Gamma(\boldsymbol{\theta})$, where the parameter λ defines the regularization strength. Common regularization practices are:

- L_2 parameter regularization,
- L_1 parameter regularization,
- Noise robustness, either by adding random perturbations to the weights or adding noise to the output layer,
- Early stopping in the gradient based optimization or annealed step size,
- Dropout, a method where only a subset of the unit weights are updated at each iteration.

As of today, it is common to use L_2 regularization combined with dropout applied after all the layers (Srivastava et al., 2014).

Neural network training

Given the objective function J, the parameters $\boldsymbol{\theta}$ of the neural network are then fitted through an optimization procedure. Due to large data volumes, this task is usually performed by mini-batch stochastic gradient descent (Bottou, 2012; Bengio, 2012). The gradient $\nabla J(\boldsymbol{\theta})$ is computed using an algorithm known as backpropagation in neural network terminology.

Forward computation (forward propagation)

We first apply forward computation through the neural network, going through all the layers from the input layer to the output layer, and compute the objective function. Let $\boldsymbol{y}$ be the ground truth classes and $\hat{\boldsymbol{y}}$ the predicted classes. We denote our loss function $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ describing how well we fit the data. To obtain the objective function J, we add a regularizer $\Gamma(\boldsymbol{\theta})$, where $\boldsymbol{\theta}$ contains all the parameters (weights and biases from all the units). For simplicity, here I describe forward propagation for only a single input example $\boldsymbol{x}$ (Algorithm 1). Practical applications should use a minibatch. The formulations are based on Goodfellow et al. (2016).

```
Algorithm 1 Forward propagation
Require: Network depth, $l$
Require: $\boldsymbol{W}^{(i)}, i \in \{1, ..., l\}$, the weight matrices of the network
Require: $\boldsymbol{b}^{(i)}, i \in \{1, ..., l\}$, the bias parameters of the network
Require: $\boldsymbol{x}$, the input to process
Require: $\boldsymbol{y}$, the target output
 1: $\boldsymbol{h}^{(0)} = \boldsymbol{x}$
 2: for $k = 1, ..., l$ do
 3:     $\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)}\boldsymbol{h}^{(k-1)}$
 4:     $\boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)})$
 5: end for
 6: $\hat{\boldsymbol{y}} = h^{(l)}$
 7: $J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda\Gamma(\boldsymbol{\theta})$
```

Backward computation (backpropagation)

Using the target $\boldsymbol{y}$, we can now derive the gradient of the objective function. Using the chain rule, we can computationally derive the gradients on the activations $\boldsymbol{a}^{(k)}$ for each layer k, starting from the output layer and going backwards to the first hidden layer. From these gradients, which can be interpreted as an indication of how each layers output should change to reduce error, one can obtain the gradient on the parameters of each layer. The gradients on weights and biases can then be immediately used as part of a stochastic gradient update or used with other gradient-based optimization methods (Algorithm 2).

```
Algorithm 2 Backpropagation
 1: After the forward propagation, compute the gradient on the output layer:
 2: $\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$
 3: for $k = l, l-1, ..., 1$ do
 4:     Convert the gradient on the layers output into a gradient into the pre- non-
        linearity activation (element-wise multiplication if f is element-wise):
 5:     $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$
 6:     Compute gradients on weights and biases (including the regularization term,
        where needed):
 7:     $\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Gamma(\boldsymbol{\theta})$
 8:     $\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\boldsymbol{h}^{(k-1)\top} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Gamma(\boldsymbol{\theta})$
 9:     Propagate the gradients w.r.t. the next lower-level hidden layers activations:
10:     $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)\top}\boldsymbol{g}$
11: end for
```

Training, validation and testing data

During the training, we optimize the weights and biases of each unit in the network. However, there are many hyper-parameters involved: the gradient step size, the regularization strength, the width and depth of the network architecture etc. For these reasons, we separate our data into three subsets: the training data, the validation data, and the testing data. The network parameters are fitted on the training set, and the hyper-parameters are tuned according to the results of the trained network on the validation data. Once the optimal hyper-parameters have been selected, and the network has been retrained on the training set, the network is applied to the testing set to measure its performance. When computationally feasible, training and validation are typically performed by cross-validation.

CONVOLUTIONAL NETWORKS

Convolutional networks are a specialized kind of feedforward network where the hidden layers perform convolution operations. First, I describe how the convolution operator is implemented in the neural network and then the derived properties. Next I present an operation called pooling, which almost all convolutional networks employ. Research into convolutional network architectures proceeds so rapidly that a new best architecture for a given benchmark is announced every few weeks to months, rendering it impractical to describe the best architecture. However, the best architectures have consistently been composed of the building blocks described here.

The convolution operator

In convolutional network terminology, inside a convolution layer, the input is convolved with a kernel. The output of this operation is sometimes referred to as a feature map. The input and the kernel are usually multi-dimensional arrays, or tensors, and the convolution operator is usually performed over more than one axis. To relate this to the feedforward networks described in the previous section, remember that discrete convolution can be viewed as multiplication with a sparse matrix.

For instance, if we use a two-dimensional image I as our input, and want to use a two-dimensional kernel K, we obtain the following output S:

$$S(i,j) = (I \star K)(i,j) = \sum_m \sum_n I(m,n)K(im,jn).$$

Convolution is commutative, meaning we can equivalently write:

$$S(i,j) = (K \star I)(i,j) = \sum_m \sum_n I(im,jn)K(m,n).$$

Usually the latter formula is more straightforward to implement in a machine learning library, because there is less variation in the range of valid values of m and n, as the kernels are typically much smaller than the input.

While the commutative property is useful for writing proofs, it is not usually an important property of a neural network implementation. Instead, it is interesting to note that many neural network libraries actually implement cross-correlation, but call it convolution:

$$S(i,j) = (I \star K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n).$$

Properties

Convolutional networks leverages three important ideas that can help improve a machine learning system: parameter sparsity, parameter sharing and equivariant representations.

Parameter sparsity

While traditional feedforward network layers use matrix multiplication by a matrix of parameters, convolutional networks typically have sparse interactions, by using kernels that are smaller than the input (Figure 3). This significantly reduces the number of stored parameters. If there are m inputs and n outputs, a matrix multiplication requires $m \times n$ parameters and the algorithms used in practice have $O(m \times n)$ runtime. If we limit the kernel size to k, then the convolution approach requires only $k \times n$ parameters and $O(k \times n)$ runtime. For many practical applications, it is possible to still obtain good performance on the machine learning task while keeping k several orders of magnitude smaller than m. Indeed, units in the deeper layers may indirectly interact with a larger portion of the input (Figure 4). This allows the network to efficiently describe complicated interactions between many variables by constructing interactions from small building blocks.

Parameters sharing

Parameter sharing refers to using the same parameter for more than one function in a model. Its is often said that a convolutional network has tied weights, because the value of the weights in the kernel applied to one input is tied to the value of the weights applied elsewhere (Figure 5). This means that rather than learning a separate set of parameters for every location, we learn only one set. This does not affect the runtime of forward propagation (which is still $O(k \times n)$ but it does further reduce the storage requirements of the model to k parameters.

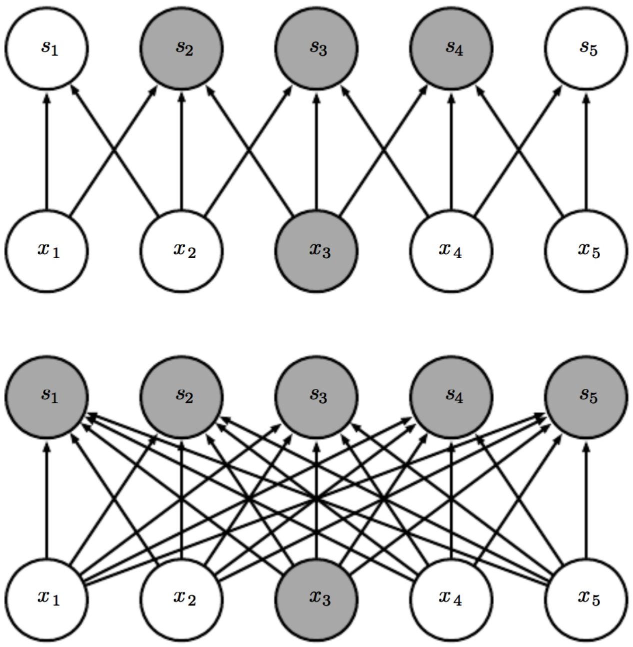

Figure 3: Parameter sparsity: We highlight one input unit, x_3, and also highlight the output units in s that are affected by this unit. (Top) When s is formed by convolution with a kernel of width 3, only three outputs are affected by x_3. (Bottom) When s is formed by full matrix multiplication, connectivity is no longer sparse, so all of the outputs are affected by x_3. (Image courtesy of Goodfelllow et al. (2016)) [**NR**]

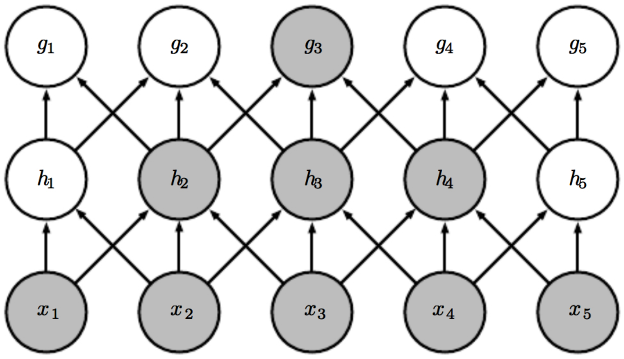

Figure 4: he receptive field of the units in the deeper layers of a convolutional network is larger than the receptive field of the units in the shallow layers. This means that even though direct connections in a convolutional network are very sparse, units in the deeper layers can be indirectly connected to all or most of the input image. (Image courtesy of Goodfelllow et al. (2016)) [**NR**]

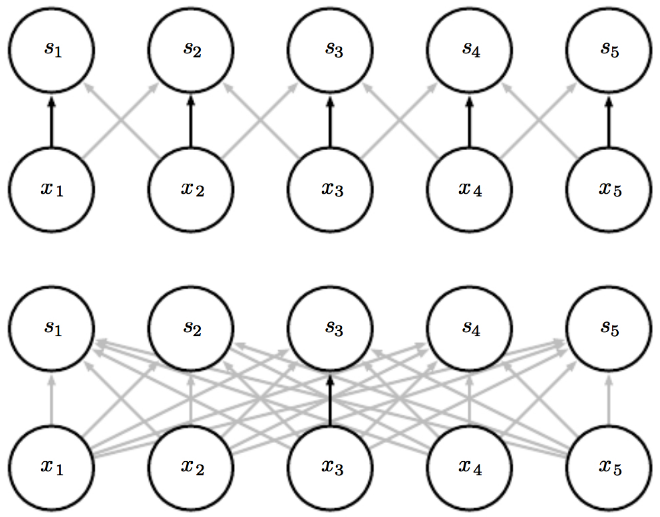

Figure 5: Parameter sharing: Black arrows indicate the connections that use a particular parameter in two different models. (Top) The black arrows indicate uses of the central element of a 3-element kernel in a convolutional model. Due to parameter sharing, this single parameter is used at all input locations. (Bottom) The single black arrow indicates the use of the central element of the weight matrix in a fully connected model. This model has no parameter sharing so the parameter is used only once. (Image courtesy of Goodfelllow et al. (2016)) [**NR**]

In practice, let's imagine a tiny image segmentation problem using a 100 × 100 grid point seismic image as input, and outputting a 100 × 100 grid point segmentation map. A traditional feedforward network with only one layer would require 10^8 operations and would have to fit 10^4 parameters. A convolutional network with a convolution stencil with 3 elements, would require 3×10^4 operations and would have to fit only 3 parameters. This shows that convolutional networks require less memory and computing time and improve statistical efficiency. In practice, however, one would use multiple stencil kernels, and the parameter space gain between a feedforward network and a convolutional network is usually of about 25-30%.

Equivariance

By using convolution with parameter sharing, the convolutional layers become equivariant to translation. This is a desirable property for processing objects with grid-type topology. For example, when processing images, it is useful to detect edges in the first layer of a convolutional network. The same edges appear more or less everywhere in the image, so it is practical to share parameters across the entire image. In some cases, we may not wish to share parameters across the entire image, for instance when trying to capture features that should be a function of a small part of the space, but there is no reason to think that the same feature should occur across all of space. In this case, one should define different kernel weights for different regions of the input space. This is sometimes called unshared convolution or locally connected layers.

Pooling

A typical layer of a convolutional network consists of three stages (Figure 6). In the first stage, the layer performs several convolutions in parallel to produce a set of linear activations. In the second stage, often called the detector stage, each linear activation is run through a nonlinear activation function, such as the rectified linear activation function. In the third stage, we use a pooling function to modify the output of the layer further.

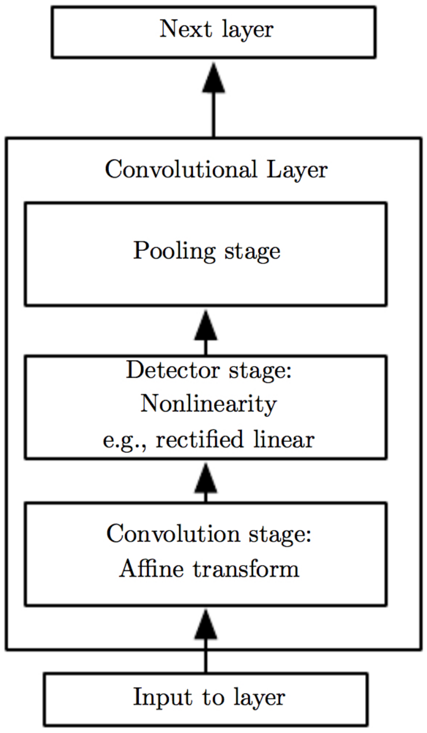

Figure 6: The components of a typical convolutional neural network layer. The convolutional network is made of several building blocks, made of a convolution stage, a non-linear activation, often called the detector stage, and a pooling stage. [**NR**]

A pooling function is a form of downsampling that replaces the output of a layer at a certain location with a summary statistic of the nearby outputs. For example, the most commonly used max pooling (Zhou and Chellappa, 1988) operation reports the maximum output within a rectangular neighborhood. Other popular pooling functions include the average of a rectangular neighborhood, the L_2 norm of a rectangular neighborhood, or a weighted average based on the distance from the central pixel. Some theoretical work gives guidance as to which kinds of pooling one should use in various situations (Boureau et al., 2010). In all cases, pooling helps to make the representation become approximately invariant to small translations of the input (Figure 7).

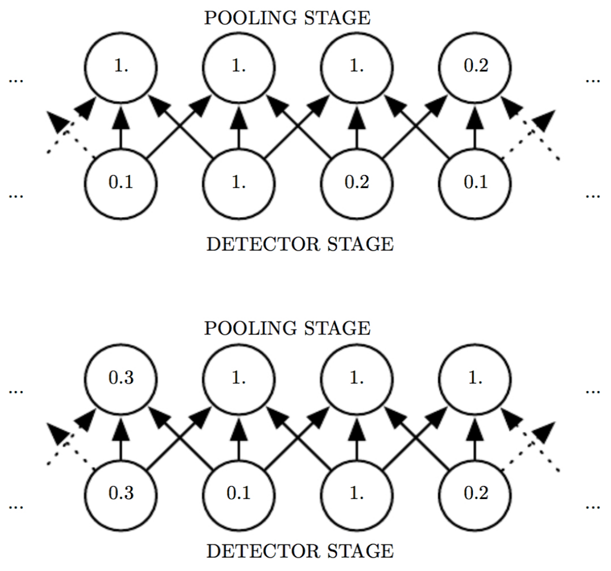

Figure 7: Max pooling introduces invariance. (Top) A view of the middle of the output of a convolutional layer. The bottom row shows outputs of the nonlinearity. The top row shows the outputs of max pooling, with a stride of one pixel between pooling regions and a pooling region width of three pixels. (Bottom) A view of the same network, after the input has been shifted to the right by one pixel. Every value in the bottom row has changed, but only half of the values in the top row have changed, because the max pooling units are only sensitive to the maximum value in the neighborhood, not its exact location. (Image courtesy of Goodfelllow et al. (2016)) [**NR**]

While pooling over spatial regions produces invariance to translation, if we pool over the outputs of separately parametrized convolutions, the features can learn which transformations to become invariant to (Figure 8). This is a strong property that allows the network to learn representative manifolds of the input data.

CONCLUSION

There are many possible convolutional network architectures to perform different tasks. Herein I described the building blocks that compose well-performing modern convolutional networks.

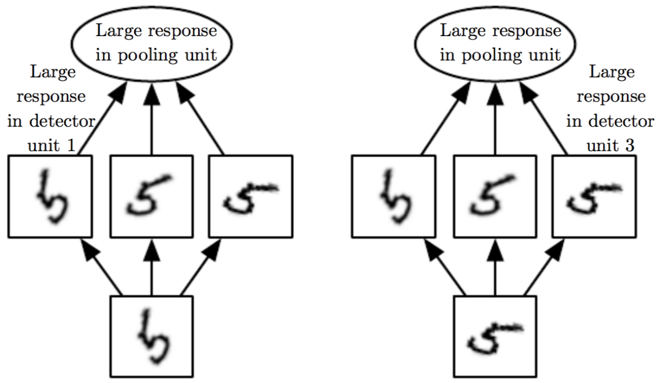
Large response in pooling unit
Large response in detector unit 1
Large response in pooling unit
Large response in detector unit 3

Figure 8: Example of learned invariances: A pooling unit that pools over multiple features that are learned with separate parameters can learn to be invariant to transformations of the input. Here we show how a set of three learned filters and a max pooling unit can learn to become invariant to rotation. All three filters are intended to detect a hand-written 5. Each filter attempts to match a slightly different orientation of the 5. When a 5 appears in the input, the corresponding filter will match it and cause a large activation in a detector unit. The max pooling unit then has a large activation regardless of which detector unit was activated. We show here how the network processes two different inputs, resulting in two different detector units being activated. The effect on the pooling unit is roughly the same either way.(Image courtesy of Goodfelllow et al. (2016)) [**NR**]

REFERENCES

Bengio, Y., 2012, Practical recommendations for gradient-based training of deep architectures, *in* Neural networks: Tricks of the trade: Springer, 437–478.

Bottou, L., 2012, Stochastic gradient descent tricks, *in* Neural networks: Tricks of the trade: Springer, 421–436.

Boureau, Y.-L., J. Ponce, and Y. LeCun, 2010, A theoretical analysis of feature pooling in visual recognition: Proceedings of the 27th international conference on machine learning (ICML-10), 111–118.

Cireşan, D., U. Meier, J. Masci, and J. Schmidhuber, 2012, Multi-column deep neural network for traffic sign classification: Neural Networks, **32**, 333–338.

Couprie, C., C. Farabet, L. Najman, and Y. LeCun, 2013, Indoor semantic segmentation using depth information: arXiv preprint arXiv:1301.3572.

Dahl, G. E., D. Yu, L. Deng, and A. Acero, 2012, Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition: IEEE Transactions on Audio, Speech, and Language Processing, **20**, 30–42.

Deng, L., M. L. Seltzer, D. Yu, A. Acero, A.-r. Mohamed, and G. E. Hinton, 2010, Binary coding of speech spectrograms using a deep auto-encoder.: Interspeech, Citeseer, 1692–1695.

Farabet, C., C. Couprie, L. Najman, and Y. LeCun, 2013, Learning hierarchical features for scene labeling: IEEE transactions on pattern analysis and machine intelligence, **35**, 1915–1929.

Goodfellow, I., Y. Bengio, and A. Courville, 2016, Deep learning: MIT Press.

Goodfellow, I. J., D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio, 2013, Maxout networks: arXiv preprint arXiv:1302.4389.

He, K., X. Zhang, S. Ren, and J. Sun, 2015, Delving deep into rectifiers: Surpassing human-level performance on imagenet classification: Proceedings of the IEEE international conference on computer vision, 1026–1034.

Hinton, G., L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, et al., 2012, Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups: IEEE Signal Processing Magazine, **29**, 82–97.

Huot, F., and R. Clapp, 2016, Detecting karst caverns by pattern recognition: SEP-Report, **163**.

Krizhevsky, A., I. Sutskever, and G. E. Hinton, 2012, Imagenet classification with deep convolutional neural networks: Advances in neural information processing systems, 1097–1105.

LeCun, Y., et al., 1989, Generalization and network design strategies: Connectionism in perspective, 143–155.

Maas, A. L., A. Y. Hannun, and A. Y. Ng, 2013, Rectifier nonlinearities improve neural network acoustic models: Presented at the Proc. ICML.

Seide, F., G. Li, and D. Yu, 2011, Conversational speech transcription using context-dependent deep neural networks.: Interspeech, 437–440.

Sermanet, P., K. Kavukcuoglu, S. Chintala, and Y. LeCun, 2013, Pedestrian detection with unsupervised multi-stage feature learning: Proceedings of the IEEE

Conference on Computer Vision and Pattern Recognition, 3626–3633.

Srivastava, N., G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, 2014, Dropout: a simple way to prevent neural networks from overfitting.: Journal of Machine Learning Research, **15**, 1929–1958.

Zhou, Y., and R. Chellappa, 1988, Computation of optical flow using a neural network: IEEE International Conference on Neural Networks, 71–78.