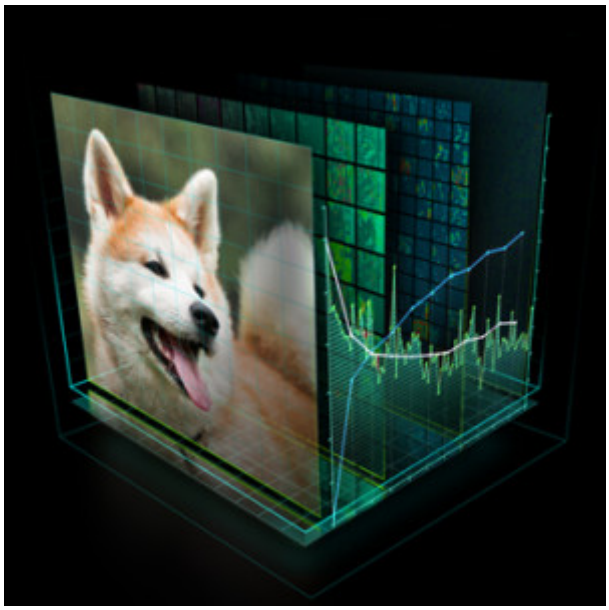
ARTIFICIAL INTELLIGENCE (HTTPS://DEVBLOGS.NVIDIA.COM/CATEGORY/ARTIFICIAL-INTELLIGENCE/)

Deep Learning in a Nutshell: Core Concepts

By Tim Dettmers (https://devblogs.nvidia.com/author/tdettmers/) | November 3, 2015 (https://devblogs.nvidia.com/deep-learning-nutshell-core-concepts/)

Tags: cuDNN (https://devblogs.nvidia.com/tag/cudnn/), Deep Learning (https://devblogs.nvidia.com/tag/deep-learning/), Deep Neural Networks (https://devblogs.nvidia.com/tag/deep-neural-networks/), Machine Learning (https://devblogs.nvidia.com/tag/machine-learning/), Neural Networks (https://devblogs.nvidia.com/tag/neural-networks/)



This post is the first in a series I'll be writing for Parallel Forall that aims to provide an intuitive and gentle introduction to deep learning (https://developer.nvidia.com/deep-learning). It covers the most important deep learning concepts and aims to provide an understanding of each concept rather than its mathematical and theoretical details. While the mathematical terminology is sometimes necessary and can further understanding, these posts use analogies and images whenever possible to provide easily digestible bits comprising an intuitive overview of the field of deep learning.

I wrote this series in a glossary style so it can also be used as a reference for deep learning concepts.

Part 1 focuses on introducing the main concepts of deep learning. Part 2 (https://devblogs.nvidia.com/parallelforall/deep-learning-nutshell-history-training/) provides historical background and delves into the training procedures, algorithms and practical tricks that are used in training for deep learning. Part 3 (https://devblogs.nvidia.com/parallelforall/deep-learning-nutshell-sequence-learning/) covers sequence learning, including recurrent neural networks, LSTMs, and encoder-decoder systems for neural machine translation. Part 4 (https://devblogs.nvidia.com/parallelforall/deep-learning-nutshell-reinforcement-learning/) covers reinforcement learning.

Core Concepts

Machine Learning

In machine learning we (1) take some data, (2) train a model on that data, and (3) use the trained model to make predictions on new data. The process of training (https://devblogs.nvidia.com/parallelforall/deep-learning-nutshell-history-training#training) a model can be seen as a learning process where the model is exposed to new, unfamiliar data step by step. At each step, the model makes predictions and gets feedback about how accurate its generated predictions were. This feedback, which is provided in terms of an error according to some measure (for example distance from the correct solution), is used to correct the errors made in prediction.

The learning process is often a game of back-and-forth in the parameter space: If you tweak a parameter of the model to get a prediction right, the model may have in such that it gets a previously correct prediction wrong. It may take many iterations to train a model with good predictive performance. This iterative predict-and-adjust process continues until the predictions of the model no longer improve.

Feature Engineering

Feature engineering is the art of extracting useful patterns from data that will make it easier for Machine Learning models to distinguish between classes. For example, you might take the number of greenish vs. bluish pixels as an indicator of whether a land or water animal is in some picture. This feature is helpful for a machine learning model because it limits the number of

classes that need to be considered for a good classification.

Feature engineering is the most important skill when you want to achieve good results for most predictions tasks. However, it is difficult to learn and master since different data sets and different kinds of data require different feature engineering approaches. Only crude guidelines exist, which makes feature engineering more of an art than a science. Features that are usable for one data set often are not usable for other data sets (for example the next image data set only contains land animals). The difficulty of feature engineering and the effort involved is the main reason to seek algorithms that can learn features; that is, algorithms that automatically engineer features.

While many tasks can be automated by Feature Learning (like object and speech recognition), feature engineering remains the single most effective technique to do well in difficult tasks (http://blog.kaggle.com/2014/08/01/learning-from-the-best/) (like most tasks in Kaggle machine learning competitions).

Feature Learning

Feature learning algorithms find the common patterns that are important to distinguish between classes and extract them automatically to be used in a classification or regression process. Feature learning can be thought of as Feature Engineering done automatically by algorithms. In deep learning, convolutional layers are exceptionally good at finding good features in images to the next layer to form a hierarchy of nonlinear features that grow in complexity (e.g. blobs, edges -> noses, eyes, cheeks -> faces). The final layer(s) use all these generated features for classification or regression (the last layer in a convolutional net is, essentially, multinomial logistic regression).

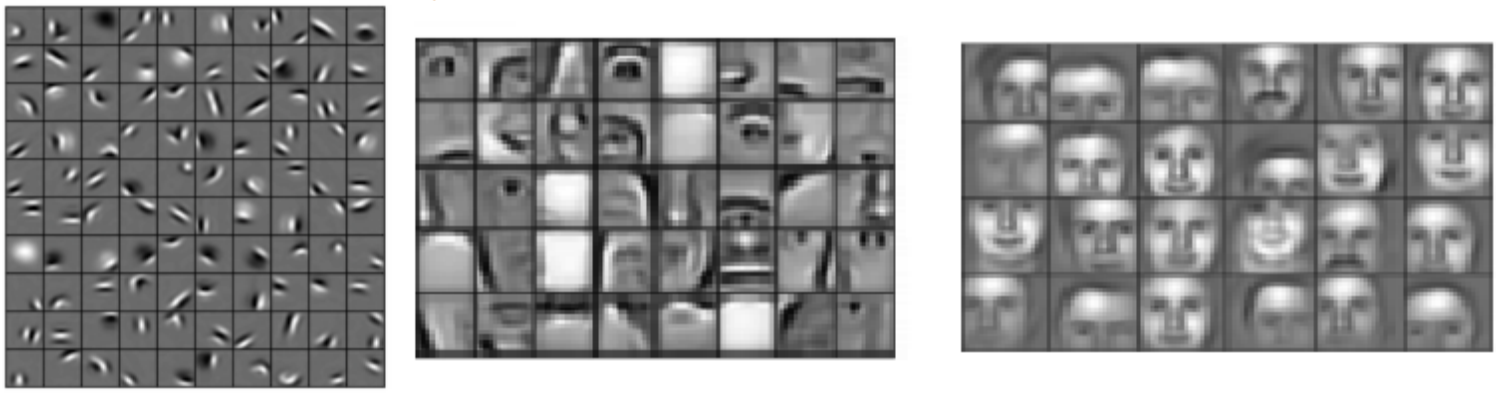


Figure 1: Learned hierarchical features from a deep learning algorithm. Each feature can be thought of as a filter, which filters the input image for that feature (a nose). If the feature is found, the responsible unit or units generate large activations, which can be picked up by the later classifier stages as a good indicator that the class is present. Image by Honglak Lee and colleagues (2011) as published in "Unsupervised Learning of Hierarchical Representations with Convolutional Deep Belief Networks".

Figure 1 shows features generated by a deep learning algorithm that generates easily interpretable features. This is rather unusual. Features are normally difficult to interpret, especially in deep networks like recurrent neural networks (https://devblogs.nvidia.com/parallelforall/deep-learning-nutshell-sequence-learning#recurrent-neural-networks) and LSTMs (https://devblogs.nvidia.com/parallelforall/deep-learning-nutshell-sequence-learning#LSTM) or very deep convolutional networks.

Deep Learning

In hierarchical Feature Learning, we extract multiple layers of non-linear features and pass them to a classifier that combines all the features to make predictions. We are interested in stacking such very deep hierarchies of non-linear features because we cannot learn complex features from a few layers. It can be shown mathematically that for images the best features for a single layer are edges and blobs because they contain the most information that we can extract from a single non-linear

transformation. To generate features that contain more information we cannot operate on the inputs directly, but we need to transform our first features (edges and blobs) again to get more complex features that contain more information to distinguish between classes.

It has been shown that the human brain does exactly the same thing: The first hierarchy of neurons that receives information in the visual cortex are sensitive to specific edges and blobs while brain regions further down the visual pipeline are sensitive to more complex structures such as faces.

While hierarchical feature learning was used before the field deep learning existed, these architectures suffered from major problems such as the vanishing gradient (https://devblogs.nvidia.com/parallelforall/deep-learning-nutshell-history-training#stochastic-gradient-descent) problem where the gradients became too small to provide a learning signal for very deep layers, thus making these architectures perform poorly when compared to shallow learning algorithms (such as support vector machines).

The term deep learning originated from new methods and strategies designed to generate these deep hierarchies of non-linear features by overcoming the problems with vanishing gradients so that we can train architectures with dozens of layers of non-linear hierarchical features. In the early 2010s, it was shown that combining GPUs with activation functions that offered better gradient flow was sufficient to train deep architectures without major difficulties. From here the interest in deep learning grew steadily.

Deep learning is not associated just with learning deep non-linear hierarchical features, but also with learning to detect very long non-linear time dependencies in sequential data. While most other algorithms that work on sequential data only have a memory of the last 10 time steps, long short-term memory (LSTM) (https://devblogs.nvidia.com/parallelforall/deep-learning-nutshell-sequence-learning#LSTM) recurrent neural networks (https://devblogs.nvidia.com/parallelforall/deep-learning-nutshell-sequence-learning#recurrent-neural-networks) (invented by Sepp Hochreiter and Jürgen Schmidhuber in 1997) allow the network to pick up on activity hundreds of time-steps in the past to make accurate predictions. While LSTM networks have been mostly ignored in the past 10 years, their usage has grown rapidly since 2013 and together with convolutional nets they form one of two major success stories of deep learning.

Fundamental Concepts

Logistic Regression

Regression analysis estimates the relationship between statistical input variables in order to predict an outcome variable. Logistic regression is a regression model that uses input variables to predict a categorical outcome variable that can take on one of a limited set of class values, for example “cancer” / “no cancer”, or an image category such as “bird” / “car” / “dog” / “cat” / “horse”.

Logistic regression applies the logistic sigmoid function (see Figure 2) to weighted input values to generate a prediction of which of two classes the input data belongs to (or in case of multinomial logistic regression, which of multiple classes).

Logistic regression is similar to a non-linear perceptron (https://devblogs.nvidia.com/parallelforall/deep-learning-nutshell-history-training#perceptron) or a neural network without hidden layers. The main difference from other basic models is that logistic regression is easy to interpret and reliable if some statistical properties for the input variables hold. If these statistical properties hold one can produce a very reliable model with very little input data. This makes logistic regression valuable for areas where data are scarce, like the medical and social sciences where logistic regression is used to analyze and interpret results from experiments. Because it is simple and fast it is also used for very large data sets.

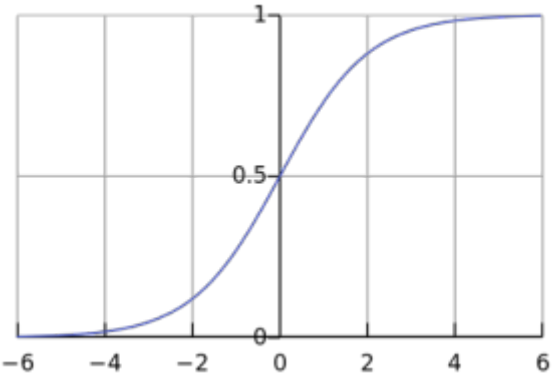


Figure 2: The logistic sigmoid function

In deep learning, the final layer of a neural network used for classification can often be interpreted as a logistic regression. In this context, one can see a deep learning algorithm as multiple feature learning stages, which then pass their features into a logistic regression that classifies an input.

Artificial Neural Network

$$f(x) = \frac{1}{1+e^{-x}}$$

. Image Source (https://en.wikipedia.org/wiki/File:Logistic-curve.svg)

An artificial neural network (1) takes some input data, and (2) transforms this input data by calculating a weighted sum over the inputs and (3) applies a non-linear function to this transformation to calculate an intermediate state. The three steps above constitute what is known as a layer, and the transformative function is often referred to as a unit. The intermediate states—often termed features—are used as the input into another layer.

Through repetition of these steps, the artificial neural network learns multiple layers of non-linear features, which it then combines in a final layer to create a prediction.

The neural network learns by generating an error signal that measures the difference between the predictions of the network and the desired values and then using this error signal to change the weights (or parameters) so that predictions get more accurate.

Unit

A unit often refers to the activation function in a layer by which the inputs are transformed via a nonlinear activation function (for example by the logistic sigmoid function (https://en.wikipedia.org/wiki/Logistic_function)). Usually, a unit has several incoming connections and several outgoing connections. However, units can also be more complex, like long short-term memory (LSTM) (https://devblogs.nvidia.com/parallelforall/deep-learning-nutshell-sequence-learning#LSTM) units, which have multiple activation functions with a distinct layout of connections to the nonlinear activation functions, or maxout units, which compute the final output over an array of nonlinearly transformed input values. Pooling, convolution, and other input transforming functions are usually not referred to as units.

Artificial Neuron

The term artificial neuron—or most often just neuron—is an equivalent term to unit, but implies a close connection to neurobiology and the human brain while deep learning has very little to do with the brain (for example, it is now thought that biological neurons are more similar to entire multilayer perceptrons rather than a single unit in a neural network). The term neuron was encouraged after the last AI winter (https://devblogs.nvidia.com/parallelforall/deep-learning-nutshell-history-training#ai-winter) to differentiate the more successful neural network from the failing and abandoned perceptron. However, since the wild successes of deep learning after 2012, the media often picked up on the term "neuron" and sought to explain deep learning as mimicry of the human brain, which is very misleading and potentially dangerous for the perception of the field of deep learning. Now the term neuron is discouraged and the more descriptive term unit should be used instead.

Activation Function

An activation function takes in weighted data (matrix multiplication between input data and weights) and outputs a non-linear transformation of the data. For example, $output = max(0, weighted_data)$ is the rectified linear activation function (https://devblogs.nvidia.com/parallelforall/deep-learning-nutshell-history-training#rectified-linear-function) (essentially set all negative values to zero). The difference between units and activation functions is that units can be more complex, that is, a unit can have multiple activation functions (for example LSTM (https://devblogs.nvidia.com/parallelforall/deep-learning-nutshell-sequence-learning#LSTM) units) or a slightly more complex structure (for example maxout units).

The difference between linear and non-linear activation functions can be shown with the relationship of some weighted values: Imagine the four points *A1*, *A2*, *B1* and *B2*. The pairs *A1* / *A2*, and *B1* / *B2* lie close to each other, but *A1* is distant from *B1* and *B2*, and vice versa; the same for *A2*.

With a linear transformation the relationship between pairs might change. For example *A1* and *A2* might be far apart, but this implies that *B1* and *B2* are also far apart. The distance between the pairs might shrink, but if it does, then both *B1* and *B2* will be close to *A1* and *A2* at the same time. We can apply many linear transformations, but the relationship between *A1* / *A2* and *B1* / *B2* will always be similar.

In contrast, with a non-linear activation function we can increase the distance between *A1* and *A2* while we *decrease* the distance between *B1* and *B2*. We can make *B1* close to *A1*, but *B2* distant from *A1*. By applying non-linear functions, we create new relationships between the points. With every new non-linear transformation we can increase the complexity of the relationships. In deep learning, using non-linear activation functions creates increasingly complex features with every layer.

In contrast, the features of 1000 layers of pure linear transformations can be reproduced by a single layer (because a chain of matrix multiplication can always be represented by a single matrix multiplication). This is why non-linear activation functions are so important in deep learning.

Layer

A layer is the highest-level building block in deep learning. A layer is a container that usually receives weighted input, transforms it with a set of mostly non-linear functions and then passes these values as output to the next layer. A layer is usually uniform, that is it only contains one type of activation function, pooling, convolution etc. so that it can be easily compared to other parts of the network. The first and last layers in a network are called input and output layers, respectively, and all layers in between are called hidden layers.

Convolutional Deep Learning

Convolution

Convolution is a mathematical operation which describes a rule of how to mix two functions or pieces of information: (1) The feature map (or input data) and (2) the convolution kernel mix together to form (3) a transformed feature map. Convolution is often interpreted as a filter, where the kernel filters the feature map for information of a certain kind (for example one kernel might filter for edges and discard other information).

Convolution is important in physics and mathematics as it defines a bridge between the spatial and time domains (pixel with intensity 147 at position (0,30)) and the frequency domain (amplitude of 0.3, at 30Hz, with 60-degree phase) through the convolution theorem. This bridge is defined by the use of Fourier transforms: When you use a Fourier transform on both the kernel and the feature map, then the convolution operation is simplified significantly (integration becomes mere multiplication). Some of the fastest GPU implementations of convolutions (for example some implementations in the NVIDIA cuDNN (https://devblogs.nvidia.com/parallelforall/accelerate-machine-learning-cudnn-deep-neural-network-library/) library) currently make use of Fourier transforms.

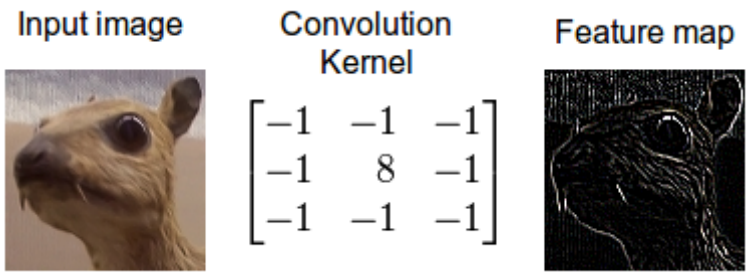


Figure 2: Convolution of an image with an edge detector convolution kernel. Sources: 1 (https://en.wikipedia.org/wiki/File:Vd-Orig.png) 2 (https://en.wikipedia.org/wiki/File:Vd-Edge3.png).

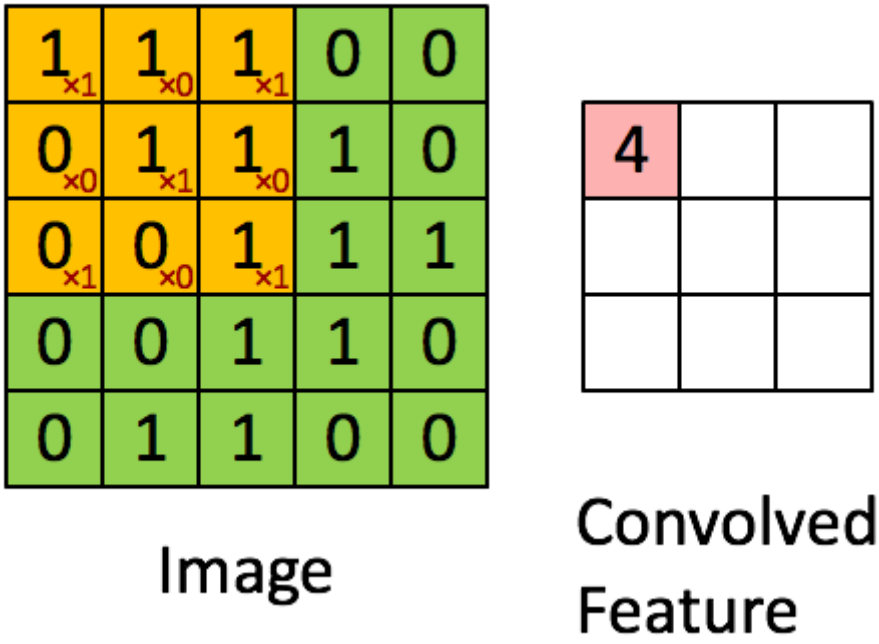


Figure 3: Calculating convolution by sliding image patches over the entire image. One image patch (yellow) of the original image (green) is multiplied by the kernel (red numbers in the yellow patch), and its sum is written to one feature map pixel (red cell in convolved feature). Image source: 1 (http://deeplearning.stanford.edu/wiki/index.php/Feature_extraction_using_convolution).

Convolution can describe the diffusion of information, for example, the diffusion that takes place if you put milk into your coffee and do not stir can be accurately modeled by a convolution operation (pixels diffuse towards contours in an image). In quantum mechanics, it describes the probability of a quantum particle being in a certain place when you measure the particle's position (average probability for a pixel's position is highest at contours). In probability theory, it describes cross-correlation, which is the degree of similarity for two sequences that overlap (similarity high if the pixels of a feature (e.g. nose) overlap in an image (e.g. face)). In statistics, it describes a weighted moving average over a normalized sequence of input (large weights for contours, small weights for everything else). Many other interpretations exist.

While it is unknown which interpretation of convolution is correct for deep learning, the cross-correlation interpretation is currently the most useful: convolutional filters can be interpreted as feature detectors, that is, the input (feature map) is filtered for a certain feature (the kernel) and the output is large if the feature is detected in the image. This is exactly how you interpret cross-correlation for an image.

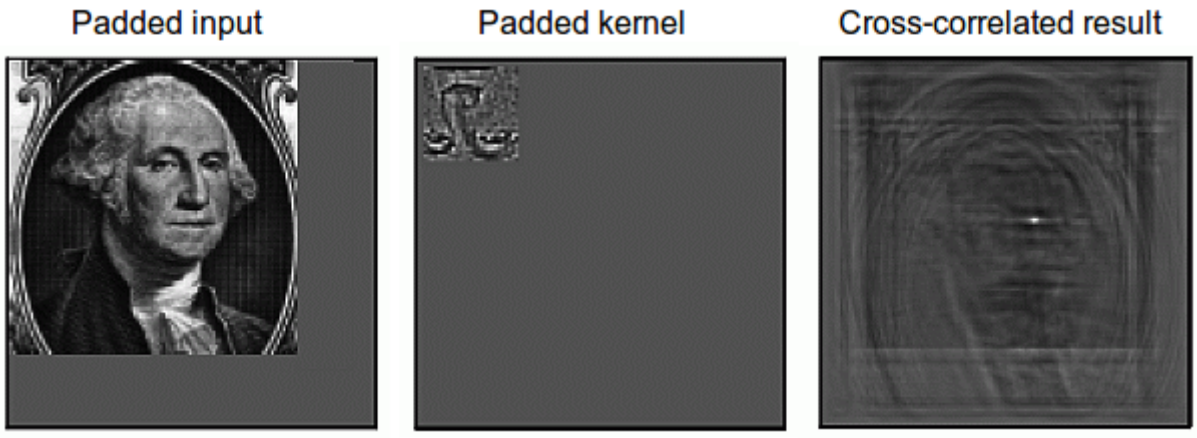


Figure 4: Cross-correlation for an image. Convolution can be transformed to cross-correlation by reversing the kernel (upside-down image). The kernel can then be interpreted as a feature detector where a detected feature results in large outputs (white) and small outputs if no feature is present (black). Images are taken from Steven Smith (http://www.dspguide.com/swsmith.htm)'s excellent free online book about digital signal processing (http://www.dspguide.com/pdfbook.htm).

Additional material: Understanding Convolution in Deep Learning (http://timdettmers.com/2015/03/26/convolution-deep-learning/)

Pooling / Subsampling

Pooling is a procedure that takes input over a certain area and reduces that to a single value (subsampling). In convolutional neural networks, this concentration of information has the useful property that outgoing connections usually receive similar information (the information is "funneled" into the right place for the input feature map of the next convolutional layer). This provides basic invariance to rotations and translations. For example, if the face on an image patch is not in the center of the image but slightly translated, it should still work fine because the information is funneled into the right place by the pooling operation so that the convolutional filters can detect the face.

The larger the size of the pooling area, the more information is condensed, which leads to slim networks that fit more easily into GPU memory. However, if the pooling area is too large, too much information is thrown away and predictive performance decreases.

Additional material: Neural networks [9.5]: Computer vision – pooling and subsampling (https://www.youtube.com/watch?v=I-JKxcpbRT4)

Convolutional Neural Network (CNN)

A convolutional neural network, or preferably convolutional network or convolutional net (the term neural is misleading; see also artificial neuron), uses convolutional layers (see convolution) that filter inputs for useful information. These convolutional layers have parameters that are learned so that these filters are adjusted automatically to extract the most useful information for the task at hand (see Feature Learning). For example, in a general object recognition task it might be most useful to filter information about the shape of an object (objects usually have very different shapes) while for a bird recognition task it might be more suitable to extract information about the color of the bird (most birds have a similar shape, but different colors; here color is more useful to distinguish between birds). Convolutional networks adjust automatically to find the best feature for these tasks.

Usually, multiple convolutional layers are used that filter images for more and more abstract information after each layer (see hierarchical features).

Convolutional networks usually also use pooling layers (see pooling) for limited translation and rotation invariance (detect the object even if it appears at some unusual place). Pooling also reduces the memory consumption and thus allows for the usage of more convolutional layers.

More recent convolutional networks use inception modules (see inception) which use 1×1 convolutional kernels to reduce the memory consumption further while speeding up the computation (and thus training).

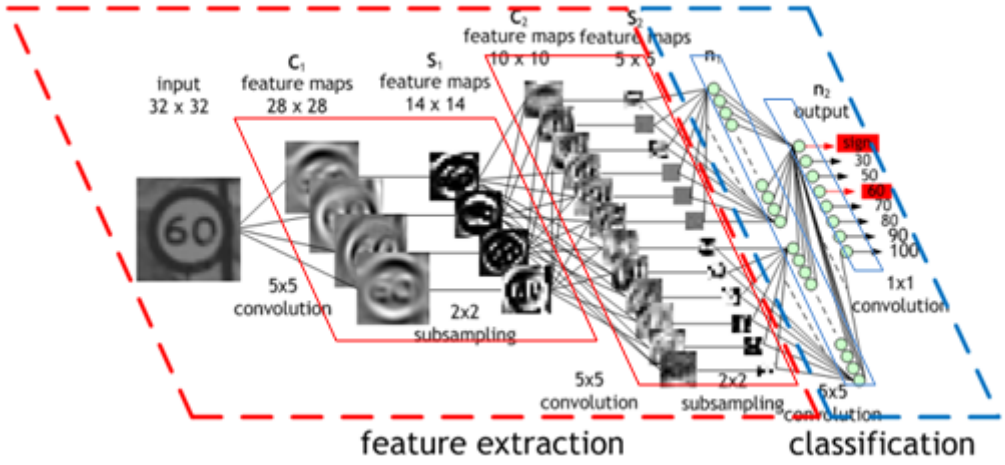


Figure 5: An image of a traffic sign is filtered by 4 5×5 convolutional kernels which create 4 feature maps, these feature maps are subsampled by max pooling. The next layer applies 10 5×5 convolutional kernels to these subsampled images and again we pool the feature maps. The final layer is a fully connected layer where all generated features are combined and used in the classifier (essentially logistic regression). Image by Maurice Peemen (http://parse.ele.tue.nl/mpeemen).

Additional material: Coursera: Neural Networks for Machine Learning: Object Recognition with Neural Nets (https://www.youtube.com/playlist?list=PLnnr1O8OWc6YLZzLoHzX2q5c2wWMOIUZy).

Inception

Inception modules in convolutional networks were designed to allow for deeper and larger convolutional layers while at the same time allowing for more efficient computation. This is done by using 1×1 convolutions with small feature map size, for example, 192 28×28 sized feature maps can be reduced to 64 28×28 feature maps through 64 1×1 convolutions. Because of the reduced size, these 1×1 convolutions can be followed up with larger convolutions of size 3×3 and 5×5. In addition to 1×1 convolution, max pooling may also be used to reduce dimensionality.

In the output of an inception module, all the large convolutions are concatenated into a big feature map which is then fed into the next layer (or inception module).

Additional material: Going Deeper with Convolutions (http://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Szegedy_Going_Deeper_With_2015_CVPR_paper.pdf)

Conclusion to Part 1

This concludes part one of this crash course on deep learning. Please check back soon for the next two parts of the series. In part 2 (https://devblogs.nvidia.com/parallelforall/deep-learning-nutshell-history-training/), I'll provide a brief historical overview followed by an introduction to training deep neural networks.

Meanwhile, you might be interested in learning about cuDNN (https://devblogs.nvidia.com/parallelforall/cudnn-v2-higher-performance-deep-learning-gpus/), DIGITS (https://devblogs.nvidia.com/parallelforall/easy-multi-gpu-deep-learning-digits-2/), Computer Vision with Caffe (https://devblogs.nvidia.com/parallelforall/deep-learning-computer-vision-caffe-cudnn/), Natural Language Processing with Torch (https://devblogs.nvidia.com/parallelforall/understanding-natural-language-deep-neural-networks-using-torch/), Neural Machine Translation (https://devblogs.nvidia.com/parallelforall/introduction-neural-machine-translation-with-gpus/), the Mocha.jl (https://devblogs.nvidia.com/parallelforall/mocha-jl-deep-learning-julia/) deep learning framework for Julia, or other Parallel Forall posts on deep learning (https://devblogs.nvidia.com/parallelforall/tag/deep-learning).

1 Comment (https://devblogs.nvidia.com/deep-learning-nutshell-core-concepts/#disqus_thread)

About the Authors

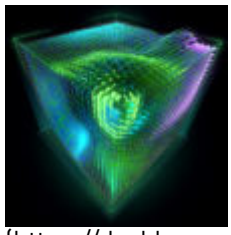
About Tim Dettmers

Tim Dettmers is a masters student in informatics at the University of Lugano where he works on deep learning research. Before that he studied applied mathematics and worked for three years as a software engineer in the automation industry. He runs a blog about deep learning (http://timdettmers.com/) and takes part in Kaggle data science competitions where he has reached a world rank of 63.
Follow @Tim_Dettmers on Twitter (https://twitter.com/intent/user?screen_name=Tim_Dettmers)
View all posts by Tim Dettmers » (https://devblogs.nvidia.com/author/tdettmers/)

Related posts

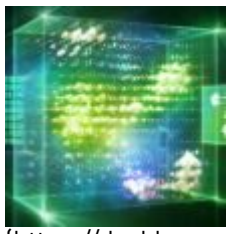
Deep Learning in a Nutshell: History and Training (https://devblogs.nvidia.com/deep-learning-nutshell-history-training/)

By Tim Dettmers (https://devblogs.nvidia.com/author/tdettmers/) | December 16, 2015 (https://devblogs.nvidia.com/deep-learning-nutshell-history-training/)


(https://devblogs.nvidia.com/dee
learning-nutshell-history-
training/)

Deep Learning in a Nutshell: Sequence Learning (https://devblogs.nvidia.com/deep-learning-nutshell-sequence-learning/)

By Tim Dettmers (https://devblogs.nvidia.com/author/tdettmers/) | March 7, 2016 (https://devblogs.nvidia.com/deep-learning-nutshell-sequence-learning/)


(https://devblogs.nvidia.com/dee
learning-nutshell-sequence-
learning/)

Deep Learning for Computer Vision with Caffe and cuDNN (https://devblogs.nvidia.com/deep-learning-computer-vision-caffe-cudnn/)

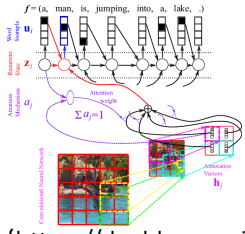
By Evan Shelhamer (https://devblogs.nvidia.com/author/eshelhammer/) | October 15, 2014 (https://devblogs.nvidia.com/deep-learning-computer-vision-caffe-cudnn/)

(https://devblogs.nvidia.com/dee

learning-computer-vision-caffe-cudnn/)

[Introduction to Neural Machine Translation with GPUs (part 3) (https://devblogs.nvidia.com/introduction-neural-machine-translation-gpus-part-3/)

By Kyunghyun Cho (https://devblogs.nvidia.com/author/kcho/) | July 26, 2015 (https://devblogs.nvidia.com/introduction-neural-machine-translation-gpus-part-3/)



(https://devblogs.nvidia.com/introneural-machine-translation-gpus-part-3/)

Comments