Deep Neural Networks: A Signal Processing Perspective

Heikki Huttunen

Abstract Deep learning has rapidly become the state of the art in machine learning, surpassing traditional approaches by a significant margin for many widely studied benchmark sets. Although the basic structure of a deep neural network is very close to a traditional 1990s style network, a few novel components enable successful training of extremely deep networks, thus allowing a completely novel sphere of applications—often reaching human-level accuracy and beyond. Below, we familiarize the reader with the brief history of deep learning and discuss the most significant milestones over the years. We also describe the fundamental components of a modern deep neural networks and emphasize their close connection to the basic operations of signal processing, such as the convolution and the Fast Fourier Transform. We study the importance of pretraining with examples and, finally, we will discuss the real time deployment of a deep network; a topic often dismissed in textbooks; but increasingly important in future applications, such as self driving cars.

1 Introduction

The research area of artificial intelligence (AI) has a long history. The first ideas of intelligent machines were raised shortly after the first computers were invented, in the 1950s. The excitement around the novel discipline with great promises led into one of the first technological hypes in computer science: In particular, military agencies such as ARPA funded the research generously, which led into a rapid expansion of the area during the 1960s and early 1970s.

As in most hype cycles, the initial excitement and high hopes were not fully satisfied. It turned out that intelligent machines able to seamlessly interact with the natural world are a lot more difficult to build than initially anticipated. This led into a

H. Huttunen (✉)
Tampere University of Technology, Tampere, Finland
e-mail: heikki.huttunen@tut.fi

S. S. Bhattacharyya et al. (eds.), *Handbook of Signal Processing Systems*,
https://doi.org/10.1007/978-3-319-91734-4_4

period of recession in artificial intelligence often called “The AI Winter”[1] during the end of 1970s. In particular, the methodologies built on top of the idea of modeling the human brain had been the most successful ones, and also the ones that suffered the most during the AI winter as the funding essentially ceased to exist for topics such as neural networks. However, the research of learning systems still continued under different names—machine learning and statistics.

The silent period of AI research was soon over, as the paradigm was refocused to study less ambitious topics than the complete human brain and seamless human-machine interaction. In particular, the rise of expert systems and the introduction of a closely related topic *data mining* led the community towards new, more focused topics.

At the beginning of 1990s the research community had already accepted that there is no silver bullet that would solve all AI problems at least in the near future. Instead, it seemed that more focused problems could be solved with tailored approaches. At the time, several successful companies had been founded, and there were many commercial uses for AI methodologies, such as the neural networks that were successful at the time. Towards the end of the century, the topic got less active and researchers directed their interest to new rising domains, such as kernel machines [35] and big data.

Today, we are in the middle of the hottest AI summer ever. Companies such as Google, Apple, Microsoft and Baidu are investing billions of dollars to AI research. Top AI conferences—such as the NIPS[2]—are rapidly sold out. The consultancy company Gartner has machine learning and deep learning at the top of their hype curve.[3] And AI even made its way to a perfume commercial.[4]

The definitive machine learning topic of the decade is *deep learning*. Most commonly, the term is used for referring to neural networks having a large number of layers (up to hundreds; even thousands). Before the current wave, neural networks were actively studied during the 1990s. Back then, the networks were significantly smaller and in particular more shallow. Adding more than two or three hidden layers only degraded the performance. Although the basic structure of a deep neural network is very close to a traditional 1990s style network, a few novel components enable successful training of extremely deep networks, thus allowing a completely novel sphere of applications—often reaching human-level accuracy and beyond.

After a silent period of the 2000s, neural networks returned to the focus of machine intelligence after Prof. Hinton from University of Toronto experimented with unconventionally big networks using unsupervised training. He discovered that training of large and deep networks was indeed possible with an unsupervised pretraining step that initializes the network weights in a layerwise manner. In the unsupervised setup, the model first learns to represent and synthesize the data

[1]https://en.wikipedia.org/wiki/History_of_artificial_intelligence.

[2]http://nips.cc/.

[3]http://www.gartner.com/newsroom/id/3784363.

[4]http://www.gq.com/story/alexandre-robicquet-ysl-model.

without any knowledge on the class labels. The second step then transforms the unsupervised model into a supervised one, and continues learning with the target labels. Another key factor to the success was the rapidly increased computational power brought by recent Graphics Processing Units (GPU's).

For a few years, different strategies of unsupervised weight initialization were at the focus of research. However, within a few years from the breakthroughs of deep learning, the unsupervised pretraining became obsolete, as new discoveries enabled direct supervised training without the preprocessing step. There is still a great interest in revisiting the unsupervised approach in order to take advantage of large masses of inexpensive unlabeled data. Currently, the fully supervised approach together with large annotated data produces clearly better results than any unsupervised approach.

The most successful application domain of deep learning is *image recognition*, which attempts to categorize images according to their visual content. The milestone event that started this line of research was the famous Alexnet network winning the annual Imagenet competition [24]. However, other areas are rising in importance, including sequence processing, such as natural language processing and machine translation.

This chapter is a brief introduction to the essential techniques behind deep learning. We will discuss the standard components of deep neural network, but will also cover some implementation topics from the signal processing perspective.

The remainder of the chapter is organized as follows. In Sect. 2, we will describe the building blocks of a modern neural network. Section 3 discusses the training algorithms and objective functions for the optimization. Finally, Sect. 4 discusses the tools for training and compares popular training platforms. We also present an example case where we compare two design strategies with examples using one of the most popular deep learning packages. Finally, Sect. 5 considers real time deployment issues in a framework where deep learning is used as one component of a system level deployment.

2 Building Blocks of a Deep Neural Network

2.1 Neural Networks

Neural networks are the core of modern artificial intelligence. Although they originally gained their inspiration from biological systems—such as the human brain—there is little in common with contemporary neural networks and their carbon-based counterparts. Nevertheless, for the sake of inspiration, let us take a brief excursion to the current understanding of the operation of biological neural networks.

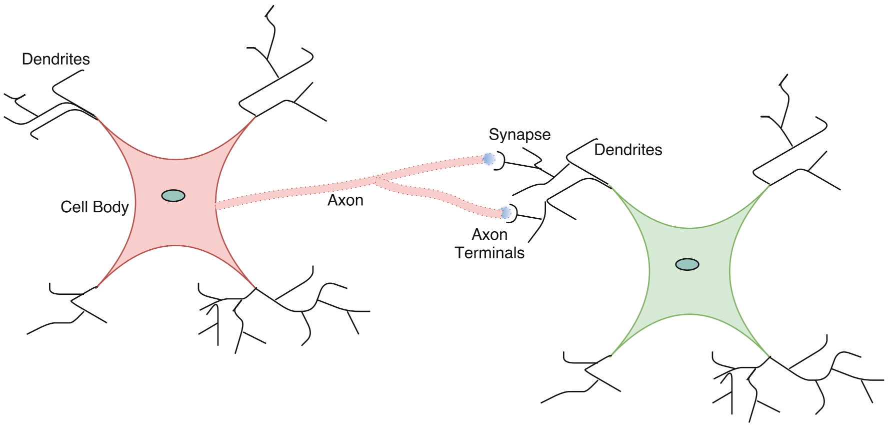

Fig. 1 A simple biological neuron network. Reprinted with permission from [45]

Figure 1 illustrates a simple biological neural network consisting of two nerve cells. The information propagates between the cells essentially through two channels: the *axon* is the transmitting terminal forwarding the level of activation to neighboring nerve cells. On the other hand, a *dendrite* serves as the receiving end, and the messages are passed through a synaptic layer between the two terminals.

Historically, the field of neural network research started in its simplest form in the 1950s, when researchers of electronics got excited about recent advances in neurology, and started to formulate the idea of an electronic brain consisting of *in silico* nerve cells that propagate their state of activity through a network of artificial cells. A landmark event of the time was the invention of Rosenblatt's *perceptron*, which uses exactly the same generalized linear model as any two-class linear classifier, such as Linear Discriminant Analysis, Logistic Regression, or the Support Vector Machine:

$$\text{Class}(\mathbf{x}) = \begin{cases} 1, & \text{if} \quad \mathbf{w}^T\mathbf{x} + b \geq 0, \\ 0, & \text{if} \quad \mathbf{w}^T\mathbf{x} + b < 0, \end{cases}$$

where $\mathbf{x} \in \mathbb{R}^N$ is the test vector to be classified, $\mathbf{w} \in \mathbb{R}^N$ and $b \in \mathbb{R}$ are the model parameters (weight vector and the bias) learned from training data. More compactly, we can write the model as $\sigma(\mathbf{w}^T\mathbf{x} + b)$ with $\sigma()$ the threshold function at zero.

The only thing differentiating the perceptron from the other generalized linear models is the training algorithm. Although not the first, nor the most powerful training algorithm, it emphasizes the idea of *iterative* learning, which presents the model training samples one at the time. Namely, the famous linear discriminant algorithm was proposed by Fisher already in the 1930s [12], but it was not used in an iterative manner due to existence of a closed form solution. When the perceptron algorithm was proposed, time was ready for exploitation of recently appeared digital computing. The training algorithm has many things in common with modern deep learning training algorithms, as well:

1. Initialize the weight vector $\mathbf{w}$ and the bias b at random.
2. For each sample $\mathbf{x}_i$ and target y_i in the training set:
   a. Calculate the model output $\hat{y}_i = \sigma(\mathbf{w}^T\mathbf{x}_i + b)$.
   b. Update the network weights by

$$\mathbf{w} := \mathbf{w} + (y_i - \hat{y}_i)\mathbf{x}_i.$$

The steps 2a and 2b correspond to the *forward pass* and *backward pass* of contemporary networks, where the samples are first propagated forward through the model to produce the output, and the error terms are pushed back as weight updates through the network in the backward pass.

For signal processing researchers, the idea of perceptron training is familiar from the field of adaptive signal processing and the Least Mean Squares (LMS) filter in particular. Coincidentally, the idea of the LMS filter was inspired by the famous Darthmouth AI meeting in 1957 [43], exactly the same year as Rosenblatt first implemented his perceptron device able to recognize a triangle held in front of its camera eye.

Fast-forwarding 30 years brings us to the introduction of the *backpropagation algorithm* [32], which enabled the training of *multilayer* perceptrons, i.e., layers of independent perceptrons stacked into a network. The structure of a 1980s multilayer perceptron is illustrated in Fig. 2. In the left, the m-dimensional input vector is fed to the first hidden layer of processing nodes (blue). Each hidden layer output is then fed to the next layer and eventually to the output layer, whose outputs are considered as class likelihoods in the classification context. The structure of each processing node is in turn illustrated in Fig. 3. Indeed, the individual neuron of Fig. 3 is very close to the 60-year-old perceptron, with a dot product followed by an activation function. This is still the exact neuron structure today, with the exception that there now exists a large library of different activations apart from the original hard thresholding, as later discussed in Sect. 2.4.

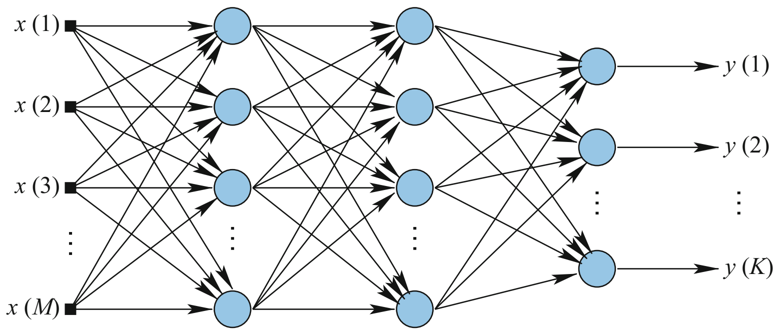

Fig. 2 A multilayer perceptron model

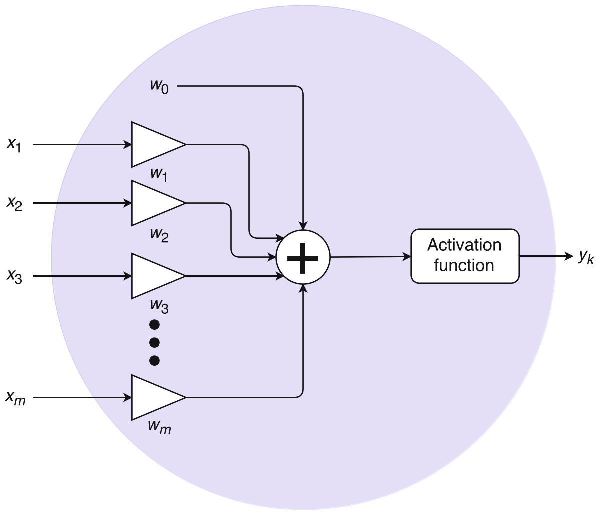

Fig. 3 A single neuron of the feedforward network of Fig. 2

2.2 Convolutional Layer

The standard layers of the 80s are today called *dense* or *fully connected* layers, reflecting their structure where *each* layer output is fed to *each* next layer node. Obviously, dense layers require a lot of parameters, which is expensive from both computational and learning point of view: High number of model coefficients require a lot of multiplications during training and deployment, but also their inference in training time is a nontrivial task. Consider, for example, the problem of image recognition (discussed later in Sect. 4.2), where we feed 64×64 RGB images into a network. If the first hidden layer were a dense layer with, say, 500 neurons, there would be over six million connections ($64 \times 64 \times 3$-dimensional input vector fed to 500 nodes requiring $500 \times 64 \times 64 \times 4 = 6{,}144{,}000$ connections). Moreover, the specific inputs would be very sensitive to geometric distortions (translations, rotations, scaling) of the input image, because each neuron can only see a single pixel at the time.

Due to these reasons, the *convolutional layer* is popular particularly in image recognition applications. As the name suggests, the convolutional layer applies a 2-dimensional convolution operation to the input. However, there are two minor differences to the standard convolution of an image processing textbook.

First, the convolution operates on multichannel input; i.e., it can see all channels of the three-channel (RGB) input image. In other words, denote the input to a convolutional layer as $\mathbf{X} \in \mathbb{R}^{M \times N \times C}$ and the convolution window as $\mathbf{W} \in \mathbb{R}^{J \times K \times C}$. Then, the output $y_{m,n}$ at spatial location (m, n) is given as

$$y_{m,n} = \sum_c \sum_j \sum_k \mathbf{W}_{j,k,c} \mathbf{X}_{m+j,n+k,c}, \tag{1}$$

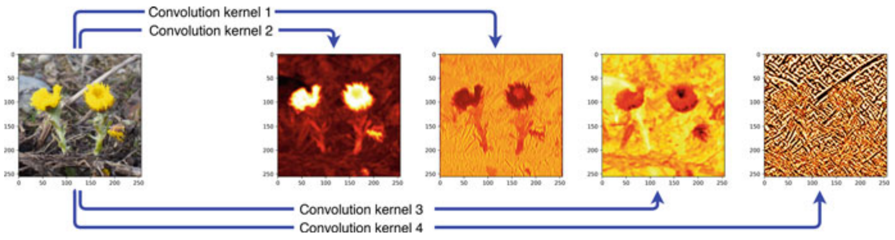

Fig. 4 Four feature maps produced from the input image by different convolution kernels

with the summation indices spanning the local window, i.e., $c = 1, 2, \ldots, C$; $j = -\lfloor \frac{J}{2} \rfloor, \ldots, \lfloor \frac{J}{2} \rfloor$ and $k = -\lfloor \frac{K}{2} \rfloor, \ldots, \lfloor \frac{K}{2} \rfloor$ assuming odd J and K. Alternatively, the convolution of Eq. (1) can also be thought of as a 3D convolution with window spanning all channels: The window only moves in the spatial dimensions, because there is no room for sliding in the channel dimension.

Second, the deep learning version of convolution does not reflect the convolution kernel with respect to the origin. Thus, we have the expression $\mathbf{X}_{m+j,n+k,c}$ in Eq. (1) instead of $\mathbf{X}_{m-j,n-k,c}$ of a standard image processing textbook. The main reason for this difference is that the weights are learned from the data, so the kernel can equally well be defined either way, and the minus is dropped out due to simplicity. Although this is a minor detail, it may cause confusion when attempting to re-implement a deep network using traditional signal processing libraries.

The role of the convolutional layer can be understood from the example of Fig. 4, where we have applied four $3 \times 3 \times 3$ convolutions to the input image. In this case, the convolution kernels highlight different features: yellow regions, green regions, diagonal edges and so on. With real convolutional networks, the kernels are learned from the data, but their role is nevertheless to extract the features essential for the application. Therefore, the outputs of the convolutions are called *feature maps* in the deep learning terminology.

In summary, the convolutional layer receives a stack of C channels (e.g., RGB), filters them with D convolutional kernels of dimension $J \times K \times C$ to produce D feature maps. Above, we considered the example with 64×64 RGB images fed to a dense layer of 500 neurons, and saw that this mapping requires 6,144,500 coefficients. For comparison, suppose we use a convolutional layer instead, producing 64 feature maps with 5×5 spatial window. In this case, each kernel can see all three channels within the local 5×5 spatial window, which requires $5 \times 5 \times 3$ coefficients. Together, all 64 convolutions are defined by $64 \times 5 \times 5 \times 3 = 4800$ parameters—over 1200 times less than for a dense layer. Moreover, the parameter count of the convolutional layer *does not depend on the image size* unlike the dense layer.

The computation of the convolution is today highly optimized using the GPU. However, the convolution can be implemented in several ways, which we will briefly discuss next. The trivial option is to compute the convolution directly using Eq. (1). However, in a close-to-hardware implementation, there are many special cases that would require specialized optimizations [6], such as small/large spatial window, small/large number of channels, or small/large number of images in batch processing essential in the training time. Although most cases can be optimized, maintaining a large number of alternative implementations soon becomes burdensome.

The second alternative is to use the Fast Fourier Transform (FFT) via the convolution theorem:

$$w(n,m) * x(n,m) = \mathrm{w.f}^{-1}\left(W(n,m)^{*} \odot X(n,m)\right), \tag{2}$$

where $W(n,m) = \mathrm{F}(w(n,m))$ and $X(n,m) = \mathrm{F}(x(n,m))$ are the discrete Fourier transforms of the convolution kernel $w(n,m)$ and one channel of the input $x(n,m)$, respectively. Moreover, F^{-1} denotes the inverse discrete Fourier transform, and * the complex conjugation that reflects the kernel about the origin in spatial domain. The obvious benefit of this approach is that the convolution is transformed to low-cost elementwise (Hadamard) product in the Fourier domain, and the computation of the FFT is faster than the convolution ($O(N \log N)$ vs. $O(N^2)$). However, the use of this approach requires that $w(n,m)$ and $x(n,m)$ are zero-padded to same size, which consumes a significant amount of temporary memory, when the filter and image sizes are far from each other. Despite these challenges, the FFT approach has shown impressive performance improvement with clever engineering [40].

The third widely used approach transforms the convolution into matrix multiplication, for which extremely well optimized implementations exist. The approach resembles the use of the classic `im2col` function in Matlab. The function rearranges the data by mapping each filter window location into a column in the result matrix. This operation is illustrated in Fig. 5, where each 3×3 block of the input (left) is vectorized into a 9×1 column of the result matrix. After this rearrangement, the convolution is simply a left multiplication with the vectorized weight matrix,

$$\mathbf{y} = \mathbf{v}^T \mathbf{C},$$

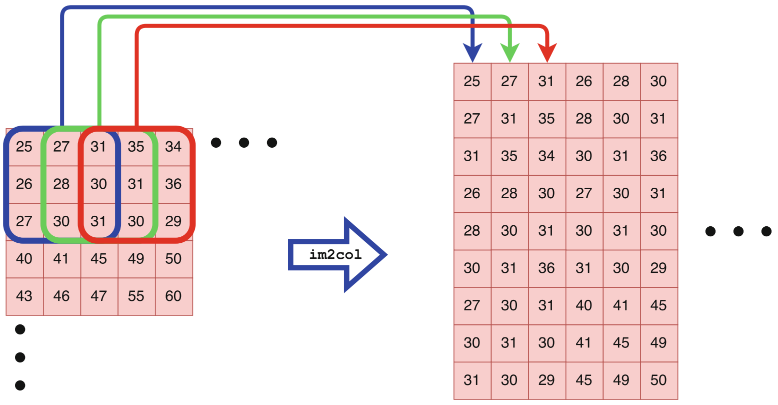

Fig. 5 The `im2col` operation

with $\mathbf{C} \in \mathbb{R}^{9\times NM}$ the result of `im2col` and $\mathbf{v} \in \mathbb{R}^{9\times 1}$ the vectorized 3×3 weight matrix. The result $\mathbf{y} \in \mathbb{R}^{1\times NM}$ can then be reshaped into to match the size of the original image. The drawback of this approach is the memory consumption of matrix $\mathbf{C}$, as the data is duplicated nine times (or more for larger window size). However, the idea provides a unified framework between convolutional layers and dense layers, since both can be implemented as matrix multiplications.

2.3 Pooling Layer

Convolutional layers are economical in terms of the number of coefficients. The parameter count is also insensitive to the size of the input image. However, the amount of *data* still remains high after the convolution, so we need some way to reduce that. For this purpose, we define the *pooling layer*, which shrinks the feature maps by some integer factor. This operation is extremely well studied in the signal processing domain, but instead of high-end decimation-interpolation process, we resort to an extremely simple approach: *max-pooling*.

Max-pooling is illustrated in Fig. 6, where we decimate the large image on the left by a factor of 2 along both spatial axes. The operation retains the largest value within each 2×2 window. Each 2×2 block is distinct (instead of sliding), so the resulting image will have half the size of the original both horizontally and vertically.

Apart from the max operation, other popular choices include taking the average, the L_2 norm, or a Gaussian-like weighted mean of the rectangular input [14, p. 330]. Moreover, the blocks may not need to be distinct, but may allow some degree of overlap. For example, [24] uses a 3×3 pooling window that strides spatially with step size 2. This corresponds to the pool window locations of Fig. 6, but the window would be extended by 1 pixel to size 3×3.

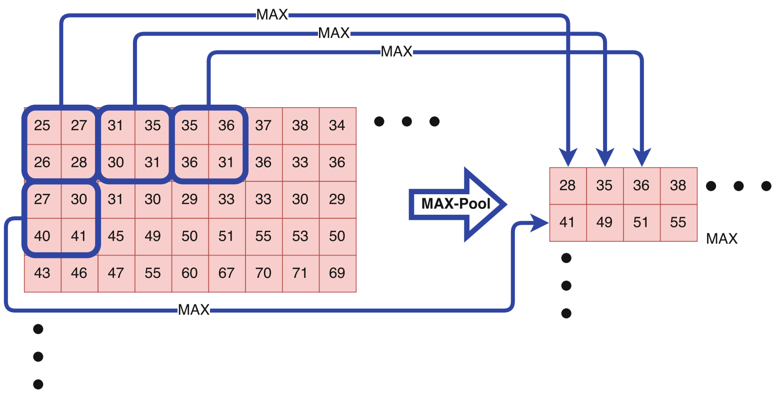

Fig. 6 The maxpooling operation

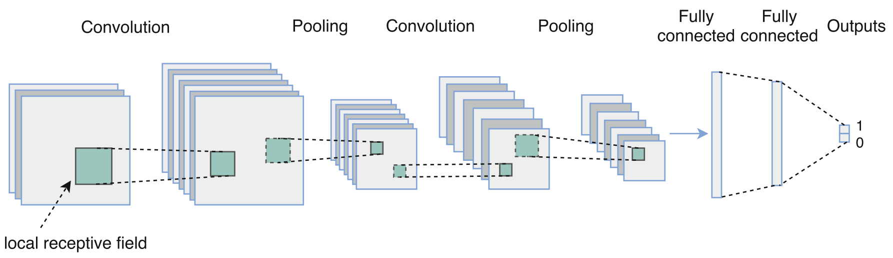

Fig. 7 Architecture of convolutional neural network (modified from [26])

The benefit of using max-pooling in particular, is its improved invariance to small translations. After the convolutions have highlighted the spatial features of interest, max-pooling will retain the largest value of the block regardless of small shifts of the input, as long as the maximum value ends up inside the local window. Translation invariance is usually preferred, because we want the same recognition result regardless of any geometric transformations.

Convolutional and pooling layers follow each other in a sequence. The convolutional layers learn to extract the essential features for the task at hand, while the pooling layers shrink the data size, together attempting to distill the essentials from the data. An example of their co-operation is illustrated in Fig. 7. In this case, the input in the left is an RGB-image (three channels). The pipeline starts with convolutions producing a number of feature maps. The feature maps are fed to the pooling layer, which shrinks each channel, but otherwise retains each map as it is. The same combination is repeated, such that the pooled feature maps are convolved with next level of convolution kernels. Note that the convolution kernel is again 3-dimensional, spanning all input channels within a small spatial window. The usual structure alternates between convolution and pooling until the amount of data is reasonable in size. At that point, the feature map channels are *flattened* (i.e., vectorized) into a vector that passes through a few dense layers. Finally, in a classification task, the number of output nodes equals the number of categories in the data; with each output interpreted as a class likelihood. As an example, a common benchmark for deep learning is the recognition of handwritten MNIST digits [25], where the inputs are 28×28 grayscale handwritten digits. In this case, there are ten classes, and the network desired output (target) is a 10-dimensional binary indicator vector—all zeros, except 1 indicating the correct class label.

2.4 Network Activations

If using only linear operations (convolution and dense layers) in a cascade, the end result could be represented by a single linear operation. Thus, the expression power will increase by introducing nonlinearities into the processing sequence; called *activation functions*. We have already seen the nonlinear thresholding operation during the discussion of the perceptron (Sect. 2.1), but the use of hard thresholding as an activation is very limited due to challenges with its gradient: The function is not differentiable everywhere and the derivative bears no information on how far we are from the origin, both important aspects for gradient based training.

Traditionally, popular network activations have been the *logistic sigmoid*,

$$\text{logsig}(x) = \frac{1}{1 + e^{-x}}, \tag{3}$$

and the *hyperbolic tangent*,

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \tag{4}$$

However, the challenge with both is that they tend to decrease the magnitude of the gradient when backpropagation is passing the weight updates through layers. Namely, the derivative of both activations is always bound to the interval $[-1, 1]$, and when backpropagation applies the chain rule, we are multiplying by a number within this range—a process that will eventually converge to zero. Thus, deep networks will encounter extremely small gradient magnitudes at the lower (close to input) layers.

While there are other approaches to circumvent this *vanishing gradient* problem, the most popular ones simply use an alternative activation function without this problem. The most widely used function is the *rectified linear unit* (ReLU) [24],

$$\text{ReLU}(x) = \max(0, x). \tag{5}$$

In other words, the function clips the input from below at zero. The benefits of the ReLU are clear: The gradient is always either 0 or 1, the computation of the function and its gradient are trivial, and experience has shown its superiority to conventional activations with many datasets. The three activation functions are illustrated in Fig. 8.

The arrangement of activation functions usually starts by setting all activations to ReLU—with the exception of output layer. The ReLU is probably not suitable for the output layer, unless our targets actually happen to fall in the range of positive reals. Common choices for output activation are either linear activation (identity mapping) in regression tasks, or logistic sigmoid in classification tasks. The sigmoid squashes the output range into the interval [0, 1], where they can be conveniently interpreted as class likelihoods. However, a more common choice is to set the ultimate nonlinearity as the *softmax* function, which additionally scales the sum of outputs $\hat{\mathbf{y}} = (\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_K)$ to unity:

$$\left[\text{softmax}(\hat{\mathbf{y}})\right]_j = \frac{\exp(\hat{y}_j)}{\sum_{k=1}^{K} \exp(\hat{y}_k)}, \quad \text{for } j = 1, 2, \ldots, K. \tag{6}$$

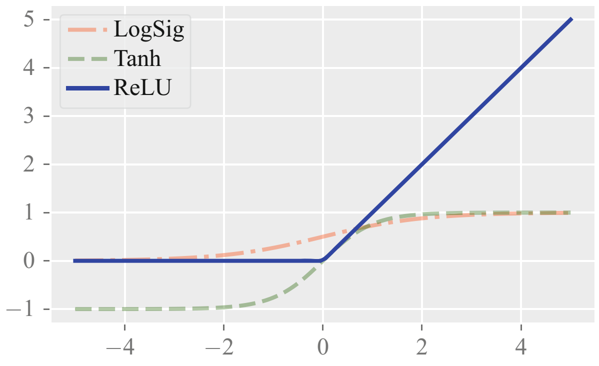

Fig. 8 Popular network activation functions

In other words, each input to the softmax layer is passed through the exponential function and normalized by their sum.

3 Network Training

The coefficients of the network layers are learned from the data by presenting examples and adjusting the weights towards the negative gradient. This process has several names: Most commonly it is called backpropagation—referring to the forward-backward flow of data and gradients—but sometimes people use the name of the optimization algorithm—the rule by which the weights are adjusted, such as *stochastic gradient descent*, *RMSProp*, *AdaGrad* [11], *Adam* [23], and so on. In [33], the good performance of backpropagation approach in several neural networks was discussed, and its importance got widely known after that. Backpropagation has two phases: propagation (forward pass) and weights update (backward pass), which we will briefly discuss next.

Forward Pass When the neural network is fed with an input, it pushes the input through the whole network until the output layer. Initially, the network weights are random, so the predictions are as good as a random guess. However, the network updates should soon push the network towards more accurate predictions.

Backward Pass Based on the prediction, the error between predictions $\hat{y}$ and target outputs y from each unit of output layer is computed using a loss function, $L(\mathbf{y}, \hat{\mathbf{y}})$, which is simply a function of the network output $\hat{\mathbf{y}} = (\hat{y}_1, \ldots, \hat{y}_N)$ and the corresponding desired targets $\mathbf{y} = (y_1, \ldots, y_N)$. We will discuss different loss functions more in detail in Sect. 3.1, but for now it suffices to note that the loss should in general be smaller when $\mathbf{y}$ and $\hat{\mathbf{y}}$ are close to each other. It is also worth noting that the network outputs are a function of the weights $\mathbf{w}$. However, in order to avoid notational clutter, we omit the explicit dependence from our notation.

Based on the loss, we compute the partial derivative of the loss $\frac{\partial L(\mathbf{y},\hat{\mathbf{y}})}{\partial \mathbf{w}}$ with respect to the weights in the network. Since the network consists of sequence of layers, the derivatives of the lower (close-to-input) layers depends on that of the upper (close-to-output) layers, and the chain rule of differentiation has to be used. A detailed discussion on how the chain rule is unrolled can be found, e.g., in [15]. Nevertheless, in order to compute the partial derivative of the loss with respect to the parameters of any of the lower layers, we need to know the derivatives of the upper layers first. Therefore, the weight update progresses from the output layer towards the input layer, which coins the name, *backpropagation*. In essence, backpropagation simply traverses the search space by updating the weights in the order admitted by the chain rule. The actual update rule then adjusts each weight towards the negative gradient with the step size specified by the parameter $\eta \in \mathbb{R}_+$:

$$w_{ij} := w_{ij} - \eta \frac{\partial L(\mathbf{y}, \hat{\mathbf{y}})}{\partial w_{ij}} \tag{7}$$

This equation is indeed exactly the same as that of the least mean square filter, familiar from adaptive signal processing.

There are various strategies for choosing the detailed weight update algorithm, as well as various possibilities for choosing the loss function $L(\mathbf{y}, \hat{\mathbf{y}})$ to be minimized. We will discuss these next.

3.1 Loss Functions

Ideally, we would like to minimize the classification error, or maximize the AUROC (area under the receiver operating characteristics curve) score, or optimize whatever quantity we believe best describes the performance of our system. However, most of these interesting performance metrics are not differentiable or otherwise intractable in closed form (for example, the derivative may not be informative enough to guide the optimization towards the optimum). Therefore, we have to use a *surrogate* target function, whose minimum matches that of our true performance metric.

Examples of commonly used loss functions are tabulated in Table 1 and plotted in Fig. 9 for a binary recognition case. In the table, we assume that the network targets $y_j \in \{0, 1\}$ for $j = 1, 2, \ldots, N$, with the exception of hinge loss, where the targets are assumed to be $y_j \in \{-1, 1\}$ for $j = 1, 2, \ldots, N$. This is the common practice in support vector machine literature where the hinge loss is most commonly used.

If we wish to maximize the classification accuracy, then our objective is to minimize the number of incorrectly classified samples. In terms of loss functions, this corresponds to the *zero-one loss* shown in Fig. 9. In this case, each network output $\hat{y}$ (a real number, higher values mean higher confidence of class membership) is rounded to the nearest integer (0 or 1) and compared to the desired target y:

Table 1 Loss functions

Loss function	Definition	Notes
Zero-one loss	$\delta\left(\langle\hat{y}\rangle, y\right)$	$\delta(\cdot,\cdot)$ is the indicator function (see text) $\langle\cdot\rangle$ denotes rounding to nearest integer
Squared loss	$\left(\hat{y}-y\right)^2$	
Absolute loss	$\left\lvert\hat{y}-y\right\rvert$	
Logistic loss	$-\ln\left(y\log(\hat{y})-(1-y)\log(1-\hat{y})\right)$	
Hinge loss	$\max\left(0, 1-y\hat{y}\right)$	Label encoding $y \in \{-1, 1\}$

In all cases, we denote the network output by $\hat{y}$ and the corresponding desired targets by y. All except hinge loss assume labels $y_j \in \{0, 1\}$ for all $j = 1, 2, \ldots, N$

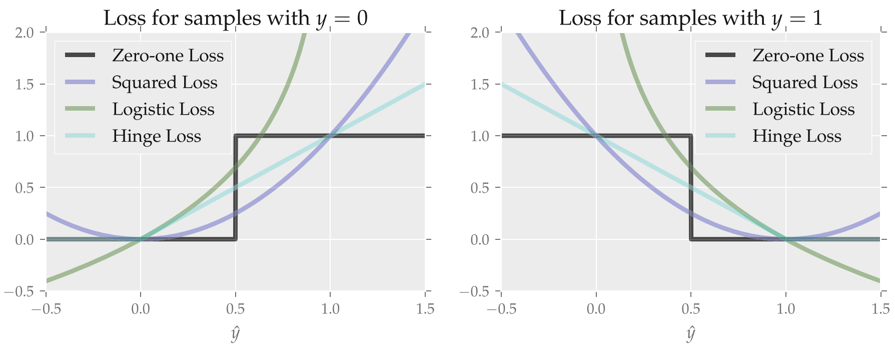

Fig. 9 Commonly used loss functions for classification. Note, that all hinge loss implementations in fact assume labels $y_j \in \{-1, 1\}$, but is scaled here to labels $y_j \in \{0, 1\}$ for visualization

$$L(\hat{y}, y) = \delta\left(\langle\hat{y}\rangle, y\right), \quad \text{with} \quad \delta(p, q) = \begin{cases} 1, & \text{if } p \neq q, \\ 0, & \text{otherwise,} \end{cases} \tag{8}$$

and $\langle x\rangle$ denotes $x \in \mathbb{R}$ rounded to the nearest integer.

Figure 9 plots selected loss functions for the two cases: $y = 0$ and $y = 1$ as a function of the network output $\hat{y}$. The zero-one loss (black) is clearly a poor target for optimization: The derivative of the loss function is zero almost everywhere and therefore conveys no information about the location of the loss minimum. Instead all of its surrogates plotted in Fig. 9 clearly direct the optimization towards the target (either 0 or 1).

In most use cases, the particular choice of loss function is less influential to the result than the optimizer used. A common choice is to use the logistic loss together with the sigmoid or softmax nonlinearity at the output.

3.2 Optimization

At training time, the network is shown labeled examples and the network weights are adjusted according to the negative gradient of the loss function. However, there are several alternative strategies on how the gradient descent is implemented.

One possibility would be to push the full training set through the network and compute the average loss over all samples. The benefit of this *batch gradient* approach would be that the averaging would give us a very stable and reliable gradient, but the obvious drawback is the resulting long waiting time until the network weights can actually be adjusted.

Similarly to the famous LMS algorithm, we obtain a similar averaging effect by using the instantaneous gradient after every sample presented to the network. This approach is called the *stochastic gradient*,

$$\mathbf{w} \leftarrow \mathbf{w} - \eta L(\hat{y}, y), \tag{9}$$

with $\eta > 0$ denoting the *step size*. Although individual gradient estimates are very noisy and may direct the optimization to a globally incorrect direction, the negative gradient will—on the average—point towards the loss minimum.

A common variant of the SGD is the so called *minibatch gradient*, which is a compromise between the batch gradient and stochastic gradient. Minibatch gradient computes the predictions (forward passes) for a *minibatch* of $B \in \mathbb{Z}_+$ samples before propagating the averaged gradient back to the lower layers:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \left(\frac{1}{B} \sum_{j=1}^{B} L(\hat{y}_j, y_j) \right). \tag{10}$$

The minibatch approach has the key benefit of speeding up the computation compared to pure SGD: A minibatch of samples can be moved to the GPU as a single data transfer operation, and the average gradient for the minibatch can be computed in a single batch operation (which parallelizes well). This will also avoid unnecessary data transfer overhead between the CPU and the GPU, which will only happen after the full minibatch is processed.

On the other hand, there is a limit to the speedup of using the minibatch. Sooner or later the GPU memory will be consumed, and the minibatch size can not be increased further. Moreover, large minibatches (up to training set size) may eventually slow down the training process, because the weight updates are happening less frequently. Although increasing the step size may compensate for this, it does not circumvent the fact that path towards the optimum may be nonlinear, and convergence would require alternating the direction by re-evaluating the local gradient more often. Thus, the sweet spot is somewhere between the stochastic gradient ($B = 1$ in Eq. (10)) and the batch gradient ($B = N$ in Eq. (10)).

Apart from the basic gradient descent, a number of improved optimization strategies have been introduced in the recent years. However, since the choice among them is nontrivial and beyond the scope of this chapter, we recommend the interested reader to study the 2012 paper by Bengio [4] or Chapter 8 of the book by Goodfellow et al. [14].

4 Implementation

4.1 Platforms

There exists several competing deep learning platforms. All the popular ones are open source and support GPU computation. They provide functionality for the basic steps of using a deep neural network: (1) Define a network model (layer structure, depth and input and output shapes), (2) train the network (define the loss function, optimization algorithm and stopping criterion), and (3) deploy the network (predict the output for test samples). Below, we will briefly discuss some of the most widely used platforms.

Caffe [21] is a deep learning framework developed and maintained by the Berkeley University Vision and Learning Center (BVLC). Caffe is written in both C++ and NVidia CUDA, and provides interfaces to Python and Matlab. The network is defined using a *Google Protocol Buffers* (prototxt) file, and trained using a command-line binary executable. Apart from the traditional manual editing of the prototxt definition file, current version also allows to define the network in Python or Matlab, and the prototxt definition will be generated automatically. A fully trained model can then be deployed either from the command line or from the Python or Matlab interfaces. Caffe is also known for the famous *Caffe Model Zoo*, where many researchers upload their model and trained weights for easy reproduction of the results in their research papers. Recently, Facebook has actively taken over the development, and released the next generation **caffe2** as open source. Caffe is licensed under the BSD license.

Tensorflow [1] is a library open sourced in 2015 by Google. Before its release, it was an internal Google project, initially under the name *DistBelief*. Tensorflow is most conveniently used through its native Python interface, although less popular C++, Java and Go interfaces exist, as well. Tensorflow supports a wide range of hardware from mobile (Android, iOS) to distributed multi-CPU multi-GPU server platforms. Easy installation packages exist for Linux, iOS and Windows through the Python *pip* package manager. Tensorflow is distributed under the Apache open source license.

Keras [7] is actually a front end for several deep learning computational engines, and links with Tensorflow, Theano [2] and Deeplearning4j backends. Microsoft is also planning to add the CNTK [44] engine into the Keras supported backends. The library is considered easy to use due to its high-level object-oriented Python interface, and it also has a dedicated *scikit-learn* API for interfacing with the extremely popular Python machine learning library [29]. The lead developer of Keras works as an engineer at Google, and it was announced that Keras will be part of the Tensorflow project since the release of Tensorflow 1.0. Keras is released under the MIT license. We will use Keras in the examples of Sect. 4.2.

Torch [9] is a library for general machine learning. Probably the most famous part of Torch is its *nn* package, which provides services for neural networks. Torch is extensively used by Facebook AI Research group, who have also released some of their own extension modules as open source. The peculiarity of Torch is its interface using *Lua* scripting language for accessing the underlying C/CUDA engine. Recently, a Python interface for Torch was released with the name **pyTorch**, which has substantially extended the user base. Torch is licensed under the BSD license.

MXNet [5] is a flexible and lightweight deep learning library. The library has interfaces for various languages: Python, R, Julia and Go, and supports distributed and multi-GPU computing. The lightweight implementation also renders it very interesting for mobile use, and the functionality of a deep network can be encapsulated into a single file for straightforward deployment into Android or iOS devices. Amazon has chosen MXNet as its deep learning framework of choice, and the library is distributed under the Apache license.

MatConvNet [41] is a Matlab toolbox for convolutional networks, particularly for computer vision applications. Although other libraries wrap their functionality into a Matlab interface, as well, MatConvNet is the only library developed as a native Matlab toolbox. On the other hand, the library can *only* be used from Matlab, as the GPU support builds on top of Matlab Parallel computing toolbox. Thus, it is the only one among our collection of platforms, that requires the purchase of proprietary software. The toolbox itself is licensed under the BSD library.

Comparison of the above platforms is challenging, as they all have their own goals. However, as all are open source projects, the activity of their user base is a critical factor predicting their future success. One possibility for estimating the popularity and the size of the community is to study the activity of their code repositories. All projects have their version control in Github development platform (http://github.com/), and one indicator of project activity is the number of *issues* raised by the users and contributors. An issue may be a question, comment or bug report, but includes also all *pull requests*, i.e., proposals for additions or changes to the project code committed by the project contributors.

The number of new issues for the above deep learning frameworks are illustrated in Fig. 10, where the curves show the number of issues per quarter since the beginning of 2015. If our crude estimate of popularity reflects the real success of each platform, then the deep learning landscape is dominated by three players: Tensorflow, Keras and the MXNet, whose combined share of issues in our graph is over 75% for Q1 of 2017.

It is also noteworthy that the pyTorch is rising its popularity very fast, although plain Torch is not. Since their key difference is the interface (Lua vs. Python), this suggests that Python has become the *de facto* language for machine learning, and every respectable platform has to provide a Python interface for users to link with their legacy Python code.

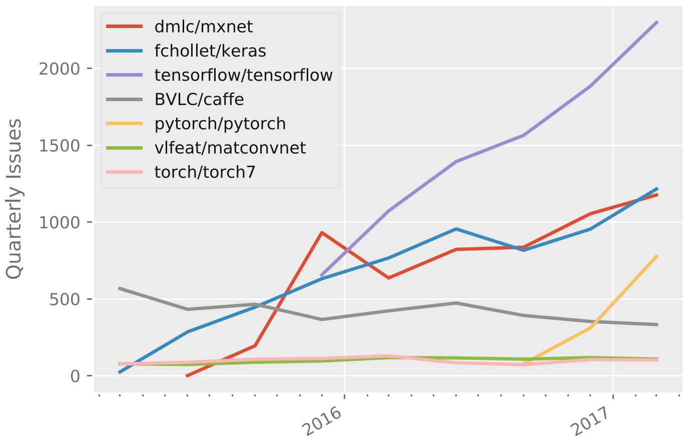

Fig. 10 Number of Github issues for popular deep learning platforms

4.2 *Example: Image Categorization*

Image categorization is probably the most studied application example of a deep learning. There are a few reasons for this. First, the introduction of the Imagenet dataset [10] in 2009 provided researchers access to a large scale heterogeneous annotated set of millions of images. Only very recently, other domains have reached data collections of equal magnitude; a recent example is the Google AudioSet database of acoustic events [13]. Large image databases were collected first, because their construction by crowdsourcing is relatively straightforward compared to, for example, annotation of audio files. The Imagenet database was collected using the Amazon Mechanical Turk crowdsourcing platform, where each user was presented an image and asked whether an object of certain category was shown in the picture. A similar human annotation for other domains is not so straightforward.

The second reason for the success of deep learning in image categorization are the ILSVRC (Internet Large Scale Visual Recognition Challenge) competitions organized annually since 2010 [34]. The challenge uses the Imagenet dataset with over one million images from 1000 categories, and different teams compete with each other in various tasks: categorization, detection and localization. The competition provides a unified framework for benchmarking different approaches, and speeds up the development of methodologies, as well. Thirdly, image recognition is a prime example of a task which is easy for humans but was traditionally difficult for machines. This raised also academic interest on whether machines can beat humans on this task.

As an example of designing a deep neural network, let us consider the Oxford Cats and Dogs dataset [28], where the task is to categorize images of *cats* and *dogs* into two classes. In the original pre-deep-learning era paper, the authors reached accuracy of 95.4% for this binary classification task. Now, let's take a look at how to

```
# Import the network container and the three types of layers
from keras.models import Sequential
from keras.layers import Conv2D, Dense, DropOut

# Initialize the model
model = Sequential ()

# Add six convolutional layers. Maxpool after every second convolution.
model.add (Conv2D (filters=32, kernel_size=3, padding=' same' , activation=' relu' ,
        input_shape =shape))
model.add (Conv2D (filters=32, kernel_size=3, padding=' same' , activation=' relu' ))
model.add (MaxPooling2D (2, 2)) # Shrink feature maps to 32x32

model.add (Conv2D(filters=48, kernel_size =3, padding =' same' , activation=' relu' ))
model.add (Conv2D(filters=48, kernel_size =3, padding =' same' , activation=' relu' ))
model.add (MaxPooling2D (2,2)) # Shrink feature maps to 16x16

model.add (Conv2D( filters=64, kernel_size=3, padding='same', activation='relu' ))
model.add (Conv2D(filters=64, kernel_size=3, padding='same', activation=' relu' ))
model.add (MaxPooling2D (2,2)) # Shrink feature maps to 8x8

# Vectorize the 8x8x64 representation to 4096x1 vector
model.add (Flatten())

# Add a dense layer with 128 nodes
model.add (Dense(128, activation='relu' ))
model.add (Dropout (0.5))

# Finally, the output layer has 1 output with logistic sigmoid nonlinearity
model.add(Dense(1, activation ='sigmoid' ))
```

Listing 1 Keras code for creating a small convolutional network with randomly initialized weights.

design a deep network using the Keras interface and how the result would compare with the above baseline.

We use a subset of 3687 images of the full dataset (1189 cats; 2498 dogs) for which the ground truth location of the animal's head is available. We crop a square shaped bounding box around the head and train the network to categorize based on this input. The bounding box is resized to fixed size 64×64 with three color channels. We choose the input size as a power of two, since it allows us to downsample the image up to six times using the maxpooling operator with stride 2.

We consider two approaches to network design:

1. Design a network from scratch,
2. Fine tune the higher layers of a pretrained network for this task.

Since the amount of training data is relatively small, the first option necessarily limits the network size in order to avoid overlearning. In the second case, the network size can be larger as it has been trained with a larger number of images before.

Small Network The structure of the network trained from scratch is shown in Fig. 11. The network consists of six convolutional layers followed by one dense

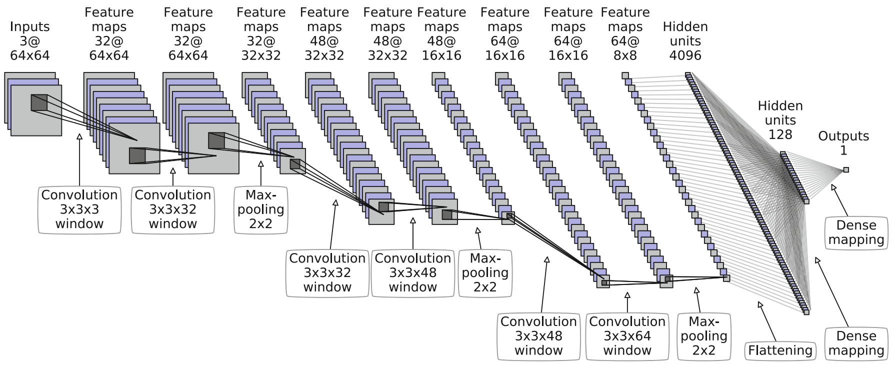

Fig. 11 Structure of the dogs and cats classification network

layer and the output layer. The input of the network is a $96 \times 96 \times 3$ array, and the output is a scalar: The probability of a dog (we encode dog as target $y_i = 1$ and cat as target $y_i = 0$). The network is created in Keras using the code on Listing 1, and the result is illustrated in Fig. 11.

The input at the left of the figure is the image to be categorized, scaled to 64×64 pixels with three color channels. The processing starts by convolving the input with a kernel with spatial size 3×3 spanning all three channels. Thus, the convolution window is in fact a cube of size $3 \times 3 \times 3$: It translates spatially along image axes, but can see all three channels at each location. This will allow the operation to highlight, e.g., all red objects by setting the red channel coefficients larger than the other channels. After the convolution operation, we apply a nonlinearity in a pixel-wise manner. In our case this is the ReLU operator: $\text{ReLU}(x) = \max(0, x)$.

Since a single convolution can not extract all the essential features from the input, we apply several of them, each with a different $3 \times 3 \times 3$ kernel. In the first layer of our example network, we decide to learn altogether 32 such kernels, each extracting hopefully relevant image features for the subsequent stages. As a result, the second layer will consist of equally many *feature maps*, i.e., grayscale image layers of size 64×64. The spatial dimensions are equal to the first layer due to the use of zero padding at the borders.

After the first convolution operation, the process continues with more convolutions. At the second layer, the $64 \times 64 \times 32$ features are processed using a convolution kernel of size $3 \times 3 \times 32$. In other words, the window has spatial dimensions 3×3, but can see all 32 channels at each spatial location. Moreover, there are again 32 such kernels, each capturing different image features from the $64 \times 64 \times 32$ image stack.

The result of the second convolution is passed to a *maxpooling* block, which resizes each input layer to 32×32—half the original size. As mentioned earlier, the shrinking is the result of retaining the largest value of each 2×2 block of each channel of the input stack. This results in a stack of 32 grayscale images of size 32×32.

The first three layers described thus far highlight the basic three-layer block that is repeated for the rest of the convolutional layer sequence. The full convolutional pipeline consists of three *convolution–convolution–maxpooling* blocks; nine layers in total. In deep convolutional networks, the block structure is very common because manual composition of a very deep network (e.g., ResNet with 152 layers [16]) or even a moderately deep network (e.g., VGG net with 16 layers [37]) is not a good target for manual design. Instead, deep networks are composed of *blocks* such as the *convolution–convolution–maxpooling* as in our case.

The network of Fig. 11 repeats the *convolution–convolution–maxpooling* block three times. After each maxpooling, we immediately increase the number of feature maps by 16. This is a common approach to avoid decreasing the data size too rapidly at the cost of reduced expression power. After the three *convolution–convolution–maxpooling* blocks, we end up with 64 feature maps of size 8×8.

The 64-channel data is next fed to two dense (fully connected) layers. To do this, we *flatten* (i.e., vectorize) the data from a $64 \times 8 \times 8$ array into a 4096-dimensional vector. This is the input to the first fully connected layer that performs the mapping

```
# Import the network container and the three types of layers
from keras.applications.vgg16 import VGG16
from keras.models import Model
from keras.layers import Conv2D

# Initialize the VGG16 network. Omit the dense layers on top.
base_model = VGG16 (include_top = False, weights = ’imagenet’ ,
                                input_shape = (64,64,3))

# We use the functional API, and grab the VGG16 output here:
w = base_model.output

# Now we can perform operations on w. First flatten it to 2048-dim vector:
w = Flatten () (w)

# Add dense layer :
w = Dense (128, activation = ’relu’ ) (w)

# Add output layer:
output = Dense (1, activation = ’sigmoid’ ) (w)

# Prepare the full model from input to output :
model = Model (input=base_model. input, output=output)

# Also set the last Conv block (3 layers) as trainable.
# There are four layers above this block, so our indices
# start at -5 (i.e., last minus five) :
model.layers [-5]. trainable = True
model.layers [-6]. trainable = True
model.layers [-7]. trainable = True
```

Listing 2 Keras code for instantiating the pretrained VGG16 network with dense layers appended on top

$\mathbb{R}^{4096} \mapsto \mathbb{R}^{128}$ by multiplying by a 128×4096-dimensional matrix followed by an elementwise ReLU nonlinearity. Finally, the result is mapped to a single probability (of a dog) by multiplying by a 1×128-dimensional matrix followed by the sigmoid nonlinearity. Note that the output is only a single probability although there are two classes: We only need one probability Prob($''$DOG$''$) as the probability of the second class is given by the complement Prob($''$CAT$''$) $= 1 -$ Prob($''$DOG$''$). Alternatively, we could have two outputs with the softmax nonlinearity, but we choose the single-output version due to its relative simplicity.

Pretrained Large Network For comparison, we study another network design approach, as well. Instead of training from scratch, we use a pretrained network which we then fine-tune for our purposes. There are several famous pretrained networks easily available in Keras, including VGG16 [37], Inception-V3 [38] and the ResNet50 [16]. All three are re-implementations of ILSVRC competition winners and pretrained weights trained with Imagenet data are available. Since the Imagenet dataset contains both cats and dogs among the 1000 classes, there is reason to believe that they should be effective for our case as well (in fact the pretrained net approach is known to be successful also for cases where the classes are not among the 1000 classes—even visually very different classes benefit from the Imagenet pretraining).

We choose the VGG16 network as our template because its 16 layers with 5 maxpoolings allow smaller input sizes than the deeper networks. The network structure follows the *convolution-convolution-maxpooling* block composition as in our own network design earlier, and is as follows.

1. **Conv block 1.** Two convolutional layers and a maxpooling layer with mapping $64 \times 64 \times 3 \mapsto 32 \times 32 \times 64$.
2. **Conv block 2.** Two convolutional layers and a maxpooling layer with mapping $32 \times 32 \times 64 \mapsto 16 \times 16 \times 128$.
3. **Conv block 3.** Three convolutional layers and a maxpooling layer with mapping $16 \times 16 \times 128 \mapsto 8 \times 8 \times 256$.
4. **Conv block 4.** Three convolutional layers and a maxpooling layer with mapping $8 \times 8 \times 256 \mapsto 4 \times 4 \times 512$.
5. **Conv block 5.** Three convolutional layers and a maxpooling layer with mapping $4 \times 4 \times 512 \mapsto 2 \times 2 \times 512$.

Additionally, the original network has three dense layers atop the five convolutional blocks. We will only use the pretrained convolutional pipeline, because the convolutional part is usually considered to serve as the feature extractor, while the dense layers do the actual classification. Therefore, the upper dense layers may be very specialized for the Imagenet problem, and would not work well in our case.

More importantly, the convolutional part is invariant to the image shape. Since we only apply convolution to the input, we can rather freely choose the input size, as long as we have large enough data to accommodate the five maxpoolings—at least 32×32 spatial size. The input shape only affects the data size at the output: for $32 \times 32 \times 3$ input we would obtain 512 feature maps of size 1×1 at the end, with

$128 \times 128 \times 3$ input the convolutional pipeline output would be of size $4 \times 4 \times 512$, and so on. The original VGG16 network was designed for $224 \times 224 \times 3$ input size, which becomes $7 \times 7 \times 512$ after five maxpooling operations. In our case the output size $2 \times 2 \times 512$ becomes 2048-dimensional vector after flattening, which is incompatible with the pretrained dense layers assuming 25,088-dimensional input.

Instead of the dense layers of the original VGG16 model, we append two layers on top of the convolutional feature extraction pipeline. These layers are exactly the same as in the small network case (see Fig. 11): One 128-node dense layer and 1-dimensional output layer. These additional layers are initialized at random.

In general, the lower layers (close to input) are less specialized to the training data than the upper layers. Since our data is not exactly similar to the Imagenet data (fewer classes, smaller spatial size, animals only), the upper convolutional layers may be less useful for us. On the other hand, the lower layers extract low level features and may be well in place for our case as well. Since our number of samples is small compared to the Imagenet data, we do not want to overfit the lower layers, but will retain them in their original state.

More specifically, we apply the backpropagation step only to the last convolutional block (and the dense layers) and keep the original pretrained coefficients for the four first convolutional blocks. In deep learning terms, we *freeze* the first four convolutional blocks. The fine-tuning should be done with caution, because the randomly initialized dense layers may feed large random gradients to the lower layers rendering them meaningless. As a rule of thumb, if in doubt, rather freeze too many layers than too few layers.

The code for instantiating the pretrained network in Keras is shown in Listing 2. Note that Keras automatically downloads the pretrained weights from the internet and keeps a local copy for the future. Listing 2 uses Keras *functional* API (in Listing 1 we used Sequential API), where each layer is defined in a functional manner, mapping the result of the previous layer by the appropriate layer type.

We train both networks with 80% of the Oxford cats and dogs dataset samples (2949 images), and keep 20% for testing (738 images). We increase the training set size by *augmentation*. Augmentation refers to various (geometric) transformations applied to the data to generate synthetic yet realistic new samples. In our case, we only use *horizontal flipping*, i.e., we reflect all training set images left-to-right. More complicated transformations would include rotation, zoom (crop), vertical flip, brightness distortion, additive noise, and so on.

The accuracy of the two network architectures is plotted in Fig. 12; on the left is the accuracy of the small network and on the right is the accuracy of the pretrained network for 50 epochs. Based on the figures, the accuracy of the pretrained network is better. Moreover, the accuracy reaches the maximum immediately after the very first epochs. The main reason for this is that the pretraining has prepared the network to produce meaningful representation for the data regardless of the input type. In essence, the pretrained classifier very close to a two layer dense network, which trains very rapidly compared to the small network with several trainable convolutional layers.

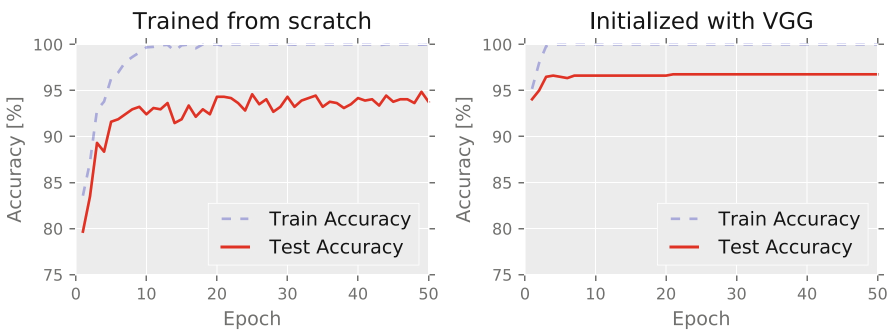

Fig. 12 The accuracy of classification for the Oxford cats and dogs dataset. Left: Learning curve of the small network initialized at random. Right: Learning curve of the fine-tuned VGG network

5 System Level Deployment

Deep learning is rarely deployed as a network only. Instead, the developer has to integrate the classifier together with surrounding software environment: Data sources, databases, network components and other external interfaces. Even in the simplest setting, we are rarely in an ideal position, where we are given perfectly cropped pictures of cats and dogs.

The *TUT Live Age Estimator* is an example of a full deep learning demo system designed to illustrate the human level abilities of a deep learning system.[5] The live video at https://youtu.be/Kfe5hKNwrCU illustrates the functionality of the demo. A screen shot of the video is also shown in Fig. 13.

The system uses three deep networks in real time:

1. An age estimator network [30, 31]
2. A gender recognizer network [30, 31]
3. An expression recognizer network

All the networks receive the cropped face, which needs to be located first. To this aim, we use the OpenCV implementation of the famous Viola-Jones object detection framework [42] with readily available face detection cascades. Moreover, the input video frames are acquired using the OpenCV VideoCapture interface.

Most of the required components are available in open source. The only thing trained by ourselves was the expression recognizer network, for which a suitable pretrained network was not available. However, after the relatively straightforward training of the one missing component, one question remains: How to put everything together?

[5]The full Python implementation is available at https://github.com/mahehu/TUT-live-age-estimator.

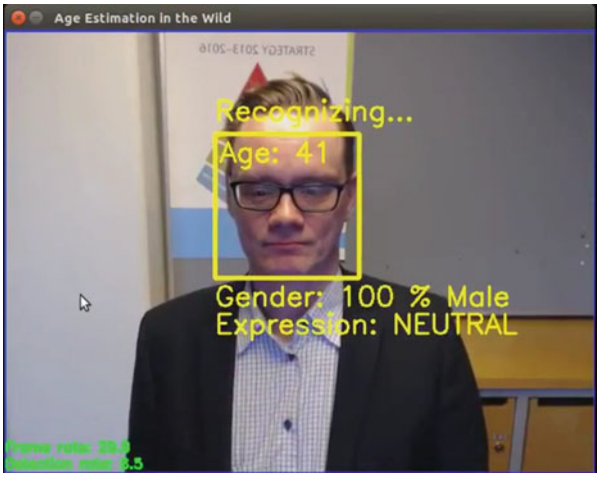

Fig. 13 Screen shot of the TUT live age estimation demo

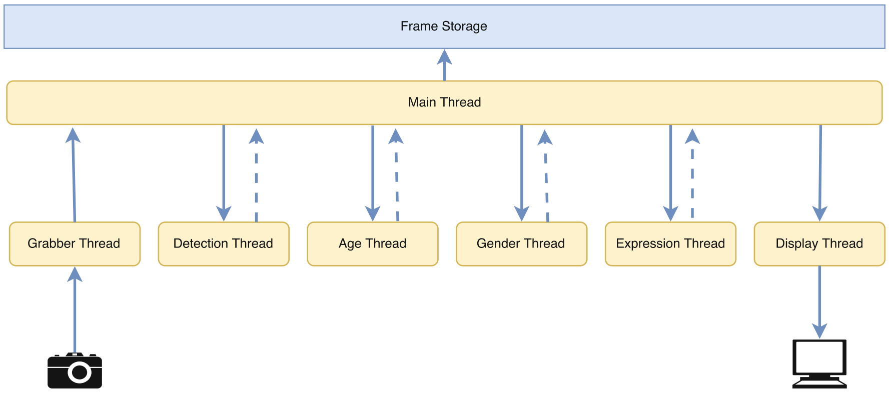

Fig. 14 Schematic diagram of the TUT age estimator

One of the challenges of the real time implementation is in the concurrency: How do we control the interplay of blocks that require different amount of computation? To this aim, we use asynchronous threads that poll for new frames to be processed. The schematic diagram of the system is shown in Fig. 14. Each stage of processing is implemented within a thread.

1. **Grabber thread** accesses the camera and requests video frames. The received frames are time stamped and pushed to the frame storage through the main thread.
2. **Detection thread** polls the frame storage for most recent frame not detected yet. When a frame is received, the OpenCV cascade classifier is applied to localize all faces. The location of the face (or *None* if not found) is added to the frame object, which also indicates that the frame has been processed.

3. **Age thread** polls the frame storage for most recent frame which has passed the detection stage but not age-recognized yet. When a frame is received, the age estimation network is applied to the cropped face. The age estimate is added to the frame object, which also indicates that the frame has been processed.
4. **Gender thread** polls the frame storage for most recent frame which has passed the detection stage but not gender-recognized yet. When a frame is received, the gender recognition network is applied to the cropped face. The gender result is added to the frame object, which also indicates that the frame has been processed.
5. **Expression thread** polls the frame storage for most recent frame which has passed the detection stage but not expression-recognized yet. When a frame is received, the expression recognition network is applied to the cropped face. The expression result is added to the frame object, which also indicates that the frame has been processed.
6. **Display thread** polls the frame storage for most recent frame not locked by any other thread for processing. The thread also requests the most recent age, gender and expression estimates and the most recent face bounding box from the main thread.
7. **Main thread** initializes all other threads and sets up the frame storage. The thread also locally keeps track of the most recent estimates of face location, age, gender and expression in order to minimize the delay of the display thread.
8. **Frame storage** is a list of frame objects. When new objects appear from the grabber thread, the storage adds the new item at the end of the list and checks whether the list is longer than the maximum allowed size. If this happens, then the oldest items are removed from the list unless locked by some processing thread. The storage is protected by mutex object to disallow simultaneous read and write.
9. **Frame objects** contain the actual video frame and its metadata, such as the timestamp, bounding box (if detected), age estimate (if recognized), and so on.

The described structure is common to many processing pipelines, where some stages are independent and allow parallel processing. In our case, the dependence is clear: Grabbing and detection are always required (in this order), but after that the three recognition events and the display thread are independent of each other and can all execute simultaneously. Moreover, if some of the processing stages needs higher priority, we can simply duplicate the thread. This will instantiate two (or more) threads each polling for frames to be processed thus multiplying the processing power.

6 Further Reading

The above overview focused on *supervised* training only. However, there are other important training modalities that an interested reader may study: *unsupervised learning* and *reinforcement learning*.

The amount of data is crucial to modern artificial intelligence. At the same time, data is often the most expensive component while training an artificial intelligent system. In particular, this is the case with annotated data used within supervised learning. Unsupervised learning attempts to learn from unlabeled samples, and the potential of unsupervised learning is not fully discovered. There is a great promise in learning from inexpensive unlabeled data instead of expensive labeled data. Not only the past of deep learning was coined by unsupervised pretraining [18, 19]; unsupervised learning may be the future of AI, as well. Namely, some of the pioneers of the field have called unsupervised learning as the future of AI,[6] since the exploitation on unlabeled data would allow exponential growth in data size.

Reinforcement learning studies problems, where the learning target consists of a sequence of operations—for example, a robot arm performing a complex task. In such cases, the entire sequence should be taken into account when defining the loss function. In other words, also the *intermediate* steps of a successful sequence should be rewarded in order to learn to solve the task successfully. A landmark paper in modern reinforcement learning is the 2015 Google DeepMind paper [27], where the authors introduce a *Deep Q-Learning* algorithm for reinforcement learning with deep neural network. Remarkably, the state-of-the-art results of the manuscript have now been obsoleted by a large margin [17], emphasizing the unprecedented speed of development in the field.

Another topic in AI with growing importance is *recurrent neural network* (RNN), which processes *sequences* using architectures that remember their past states. This enables the concept of *memory*, which allows storage of either temporal or otherwise sequential events for future decisions. Recurrent networks can have multiple configurations depending on the problem input/output characteristics, and Fig. 15 illustrates a few common ones. The particular characteristic of a recurrent network is that it can process sequences with applications such as *image captioning* [22], *action recognition* [36] or *machine translation* [3]. The most widely used RNN structures include the *Long Short-Term Memory* (LSTM) networks [20] and *Gated Recurrent Unit* (GRU) networks [8].

7 Conclusions

Deep learning has become a standard tool in any machine learning practitioner's toolbox surprisingly fast. The power of deep learning resides in the layered structure, where the early layers distill the essential features from the bulk of data, and the upper layers eventually classify the samples into categories. The research on the field is extremely open, with all important papers openly publishing their code along with the submission. Moreover, the researchers are increasingly aware of the

[6]http://spectrum.ieee.org/automaton/robotics/artificial-intelligence/facebook-ai-director-yann-lecun-on-deep-learning.

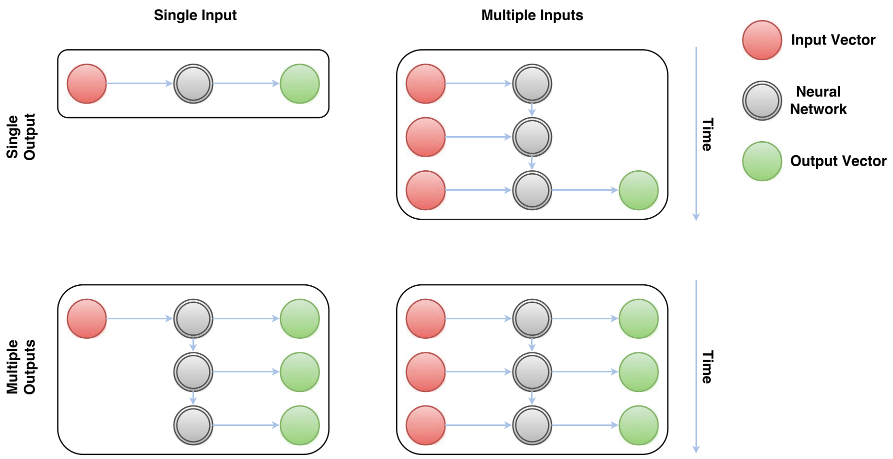

Fig. 15 Configurations of recurrent neural networks. *Top left:* A non-recurrent network, with a single input (e.g., facial image) and single output (e.g., age). *Top right:* A recurrent network with sequence input (e.g., video frames) and a single output (e.g., action of the user in the sequence). *Bottom left:* A recurrent network with single input (e.g., an image) and a sequence output (e.g., the image caption text). *Bottom right:* a recurrent network with a sequence input (e.g. text in Swedish) and sequence output (e.g., text in English)

importance of publishing open access; either in gold open access journals or via preprint servers, such as the ArXiv. The need for this kind of reproducible research was noted early in the signal processing community [39] and has luckily become the standard operating principle of machine learning.

The remarkable openness of the community has led to democratization of the domain: Today everyone can access the implementations, the papers, and other tools. Moreover, cloud services have brought also the hardware accessible to almost everyone: Renting a GPU instance from Amazon cloud, for instance, is affordable. Due to the increased accessibility, standard machine learning and deep learning have become a bulk commodity: Increased number of researchers and students possess the basic abilities in machine learning. So what's left for research, and where the future will lead us?

Despite the increased supply of experts, also the demand surges due to the growing business in the area. However, the key factors of tomorrow's research are twofold. First, *data* will be the currency of tomorrow. Although large companies are increasingly open sourcing their code, they are very sensitive to their business critical data. However, there are early signs that this may change, as well. Companies are opening their data as well: One recent surprise was the release of Google AudioSet—a large-scale dataset of manually annotated audio events [13]—which completely transformed the field of sound event detection research.

Second, the current wave of deep learning success has concentrated on the virtual world. Most of the deep learning is done in server farms using data from the cloud. In other words, the connection to the *physical world* is currently very slim. This is about to change; as an example, deep learning is rapidly steering the design of self driving cars, where the computers monitor their surroundings via dashboard mounted cameras. However, most of the current platforms are at a prototype stage, and we will see more application-specific deep learning hardware in the future. We have also seen that most of the deep learning computation operations stem from basic signal processing algorithms, embedded DSP design expertise may be in high demand in the coming years.

Acknowledgements The author would like to acknowledge *CSC - IT Center for Science Ltd.* for computational resources.

References

1. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., et al.: Tensorflow: Large-scale machine learning on heterogeneous distributed systems. arXiv preprint arXiv:1603.04467 (2016)
2. Al-Rfou, R., Alain, G., Almahairi, A., Angermueller, C., Bahdanau, D., Ballas, N., Bastien, F., Bayer, J., Belikov, A., Belopolsky, A., et al.: Theano: A python framework for fast computation of mathematical expressions. arXiv preprint arXiv:1605.02688 (2016)
3. Bahdanau, D., Cho, K., Bengio, Y.: Neural machine translation by jointly learning to align and translate. Proceedings of ICLR2015 (2015)
4. Bengio, Y.: Practical recommendations for gradient-based training of deep architectures. In: Neural networks: Tricks of the trade, pp. 437–478. Springer (2012)
5. Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., Zhang, Z.: Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. arXiv preprint arXiv:1512.01274 (2015)
6. Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., Shelhamer, E.: cudnn: Efficient primitives for deep learning. arXiv preprint arXiv:1410.0759 (2014)
7. Chollet, F.: Keras. https://github.com/fchollet/keras (2015)
8. Chung, J., Gulcehre, C., Cho, K., Bengio, Y.: Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. Proceedings of NIPS conference (2014)
9. Collobert, R., Kavukcuoglu, K., Farabet, C.: Torch7: A matlab-like environment for machine learning. In: BigLearn, NIPS Workshop (2011)
10. Deng, J., Dong, W., Socher, R., Li, L.J., Li, K., Fei-Fei, L.: ImageNet: A Large-Scale Hierarchical Image Database. In: CVPR09 (2009)
11. Duchi, J., Hazan, E., Singer, Y.: Adaptive subgradient methods for online learning and stochastic optimization. Journal of Machine Learning Research **12**(Jul), 2121–2159 (2011)
12. Fisher, R.A.: The use of multiple measurements in taxonomic problems. Annals of eugenics **7**(2), 179–188 (1936)
13. Gemmeke, J.F., Ellis, D.P.W., Freedman, D., Jansen, A., Lawrence, W., Moore, R.C., Plakal, M., Ritter, M.: Audio set: An ontology and human-labeled dataset for audio events. In: Proc. IEEE ICASSP 2017. New Orleans, LA (2017)
14. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press (2016). http://www.deeplearningbook.org
15. Haykin, S., Network, N.: A comprehensive foundation. Neural Networks **2**(2004), 41 (2004)
16. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 770–778 (2016). https://doi.org/10.1109/CVPR.2016.90

17. Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., Silver, D.: Rainbow: Combining Improvements in Deep Reinforcement Learning. ArXiv e-prints (2017). Submitted to AAAI2018
18. Hinton, G.E.: Learning multiple layers of representation. Trends in Cognitive Sciences **11**(10), 428–434 (2007)
19. Hinton, G.E., Osindero, S., Teh, Y.W.: A fast learning algorithm for deep belief nets. Neural Computation **18**(7), 1527–1554 (2006)
20. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural computation **9**(8), 1735–1780 (1997)
21. Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., Darrell, T.: Caffe: Convolutional architecture for fast feature embedding. In: Proceedings of the ACM International Conference on Multimedia, pp. 675–678. ACM (2014)
22. Karpathy, A., Fei-Fei, L.: Deep visual-semantic alignments for generating image descriptions. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 3128–3137 (2015)
23. Kingma, D., Ba, J.: Adam: A method for stochastic optimization. International Conference on Learning Representations (2015)
24. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: F. Pereira, C.J.C. Burges, L. Bottou, K.Q. Weinberger (eds.) Advances in Neural Information Processing Systems 25, pp. 1097–1105. Curran Associates, Inc. (2012)
25. LeCun, Y., Boser, B., Denker, J.S., Henderson, D., Howard, R.E., Hubbard, W., Jackel, L.D.: Backpropagation applied to handwritten zip code recognition. Neural computation **1**(4), 541–551 (1989)
26. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE pp. 2278–2324 (1998)
27. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., et al.: Human-level control through deep reinforcement learning. Nature **518**(7540), 529–533 (2015)
28. Parkhi, O.M., Vedaldi, A., Zisserman, A., Jawahar, C.V.: Cats and dogs. In: IEEE Conference on Computer Vision and Pattern Recognition (2012)
29. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in Python. Journal of Machine Learning Research **12**, 2825–2830 (2011)
30. Rothe, R., Timofte, R., Gool, L.V.: Dex: Deep expectation of apparent age from a single image. In: IEEE International Conference on Computer Vision Workshops (ICCVW) (2015)
31. Rothe, R., Timofte, R., Gool, L.V.: Deep expectation of real and apparent age from a single image without facial landmarks. International Journal of Computer Vision (IJCV) (2016)
32. Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning representations by back-propagating errors. Nature **323**(6088), 533–538 (1986)
33. Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning representations by back-propagating errors. Cognitive modeling **5**(3), 1 (1988)
34. Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A.C., Fei-Fei, L.: ImageNet Large Scale Visual Recognition Challenge. International Journal of Computer Vision (IJCV) **115**(3), 211–252 (2015). https://doi.org/10.1007/s11263-015-0816-y
35. Schölkopf, B., Smola, A.J.: Learning with kernels. The MIT Press (2001)
36. Simonyan, K., Zisserman, A.: Two-stream convolutional networks for action recognition in videos. In: Advances in neural information processing systems, pp. 568–576 (2014)
37. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 (2014)
38. Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., Wojna, Z.: Rethinking the inception architecture for computer vision. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 2818–2826 (2016)

39. Vandewalle, P., Kovacevic, J., Vetterli, M.: Reproducible research in signal processing. IEEE Signal Processing Magazine **26**(3) (2009)
40. Vasilache, N., Johnson, J., Mathieu, M., Chintala, S., Piantino, S., LeCun, Y.: Fast convolutional nets with fbfft: A gpu performance evaluation. arXiv preprint arXiv:1412.7580 (2014)
41. Vedaldi, A., Lenc, K.: Matconvnet – convolutional neural networks for matlab. In: Proceeding of the ACM Int. Conf. on Multimedia (2015)
42. Viola, P., Jones, M.: Rapid object detection using a boosted cascade of simple features. In: Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on, vol. 1, pp. I–I. IEEE (2001)
43. Widrow, B.: Thinking about thinking: the discovery of the lms algorithm. IEEE Signal Processing Magazine **22**(1), 100–106 (2005). https://doi.org/10.1109/MSP.2005.1407720
44. Yu, D., Eversole, A., Seltzer, M., Yao, K., Huang, Z., Guenter, B., Kuchaiev, O., Zhang, Y., Seide, F., Wang, H., et al.: An introduction to computational networks and the computational network toolkit. Microsoft Technical Report MSR-TR-2014–112 (2014)
45. Zhu, L.: Gene expression prediction with deep learning. M.Sc. Thesis, Tampere University of Technology (2017)