Everything you wanted to know about Deep Learning for Computer Vision but were afraid to ask

Moacir A. Ponti, Leonardo S. F. Ribeiro, Tiago S. Nazare
ICMC – University of São Paulo
São Carlos/SP, 13566-590, Brazil
Email: [ponti, leonardo.sampaio.ribeiro, tiagosn]@usp.br

Tu Bui, John Collomosse
CVSSP – University of Surrey
Guildford, GU2 7XH, UK
Email: [t.bui, j.collomosse]@surrey.ac.uk

***Abstract*—Deep Learning methods are currently the state-of-the-art in many Computer Vision and Image Processing problems, in particular image classification. After years of intensive investigation, a few models matured and became important tools, including Convolutional Neural Networks (CNNs), Siamese and Triplet Networks, Auto-Encoders (AEs) and Generative Adversarial Networks (GANs). The field is fast-paced and there is a lot of terminologies to catch up for those who want to adventure in Deep Learning waters. This paper has the objective to introduce the most fundamental concepts of Deep Learning for Computer Vision in particular CNNs, AEs and GANs, including architectures, inner workings and optimization. We offer an updated description of the theoretical and practical knowledge of working with those models. After that, we describe Siamese and Triplet Networks, not often covered in tutorial papers, as well as review the literature on recent and exciting topics such as visual stylization, pixel-wise prediction and video processing. Finally, we discuss the limitations of Deep Learning for Computer Vision.**

***Keywords*-Computer Vision; Deep Learning; Image Processing; Video Processing**

I. Introduction

The field of Computer Vision was revolutionized in the past years (in particular after 2012) due to Deep Learning techniques. This is mainly due to two reasons: the availability of labelled image datasets with millions of images [1], [2], and computer hardware that allowed to speed-up computations. Before that, there were studies exploring hierarchical representations with neural networks such as Fukushima's Neocognitron [3] and LeCun's neural networks for digit recognition [4]. Although those techniques were known to Machine Learning and Artificial Intelligence communities, the efforts of Computer Vision researchers during the 2000's was in a different direction, in particular using approaches based on Scale-Invariant features, Bag-of-Features, Spatial Pyramids and related methods [5].

After the publication of the AlexNet Convolutional Neural Network Model [6], many research communities realized the power of such methods and, from that point on, Deep Learning (DL) invaded the Visual Computing fields: Computer Vision, Image Processing, Computer Graphics. Convolutional Neural Networks (CNNs), Deep Belief Nets (DBNs), Restricted Boltzmann Machines (RBMs) and Autoencoders (AEs), started appearing as a basis for state-of-the-art methods in several computer vision applications (e.g. remote sensing [7], surveillance [8], [9] and re-identification [10]). The ImageNet challenge [1] played a major role in this process, starting a race for the model that could beat the current champion in the image classification challenge, but also image segmentation, object recognition and other tasks. In order to accomplish that, different architectures and combinations of DBM were employed. Also, novel types of networks – such as Siamese Networks, Triplet Networks and Generative Adversarial Networks (GANs) – were designed.

Deep Learning techniques offer an important set of methods suited for tasks within the domains of digital visual content. It is noticeable that those methods comprise a diverse variety of models, components and algorithms that may be dissimilar to what one may be used to in Computer Vision. A myriad of keywords within this context makes the literature in this field almost a different language: Feature maps, Activation, Receptive Fields, Dropout, ReLu, Max-Pool, Softmax, SGD, Adam, FC, Generator, Discriminator, Shared Weights, etc. This can make it hard for a beginner to understand and catch up with the recent studies.

There are DL methods we do not cover in this paper, including Deep Belief Networks (DBN), Deep Boltzmann Machines (DBM) and also those using Recurrent Neural Networks (RNN), Reinforcement Learning and Long short-term memory (LSTMs). We refer to [11]–[14] for DBN and DBM-related studies, and [15]–[17] for RNN-related studies.

The paper is organized as follows:

- Section II provides **definitions and prerequisites**.
- Section III aims to present a detailed and updated description of the DL's main terminology, building blocks and algorithms of the **Convolutional Neural Network (CNN)** since it is widely used in Computer Vision, including:
  1) *Components*: Convolutional Layer, Activation Function, Feature/Activation Map, Pooling, Normalization, Fully Connected Layers, Parameters, Loss Function;
  2) *Algorithms*: Optimization (SGD, Momentum,

Adam, etc.) and Training;
3) *Architectures*: AlexNet, VGGNet, ResNet, Inception, DenseNet;
4) *Beyond Classification*: fine-tuning, feature extraction and transfer learning.

- Section IV describes **Autoencoders (AEs)**:
  1) *Undercomplete AEs*;
  2) *Overcomplete AEs*: Denoising and Contractive;
  3) *Generative AE*: Variational AEs
- Section V introduces **Generative Adversarial Networks (GANs)**:
  1) *Generative Models*: Generator/Discriminator;
  2) *Training*: Algorithm and Loss Lunctions.
- Section VI is devoted to **Siamese and Triplet Networks**:.
  1) *SiameseNet* with contrastive Loss;
  2) *TripletNet*: with triplet Loss;
- Section VII reviews **Applications of Deep Networks in Image Processing and Computer Vision**, including:
  1) *Visual Stilization*;
  2) *Image Processing and Pixel-Wise Prediction*;
  3) *Networks for Video Data*: Multi-stream and C3D.
- Section VIII concludes the paper by discussing the **Limitations of Deep Learning** methods for Computer Vision.

II. Deep Learning: prerequisites and definitions

The prerequisites needed to understand Deep Learning for Computer Vision includes basics of Machine Learning (ML) and Image Processing (IP). Since those are out of the scope of this paper we refer to [18], [19] for an introduction in such fields. We also assume the reader is familiar with Linear Algebra and Calculus, as well as Probability and Optimization — a review of those topics can be found in Part I of Goodfellow *et al*. textbook on Deep Learning [20].

Machine learning methods basically try to discover a model (e.g. rules, parameters) by using a set of input data points and some way to guide the algorithm in order to learn from this input. In supervised learning, we have examples of expected output, whereas in unsupervised learning some assumption is made in order to build the model. However, in order to achieve success, it is paramount to have a good representation of the input data, i.e. a good set of features that will produce a feature space in which an algorithm can use its bias in order to learn. One of the main ideas of **Deep learning** is to solve the problem of finding this representation by learning it from the data: it defines representations that are expressed in terms of other, simpler ones [20]. Another is to assume that depth (in terms of successive representations) allows learning a sequence of parallel instructions that transforms an initial input vector, mapping one space to another.

Deep Learning often involves learning hierarchical representations using a series of layers that operate by processing an input generating a series of representations that are then given as input to the next layer. By having spaces of sufficiently high dimensionality, such methods are able to capture the scope of the relationships found in the original data so that it finds the "right" representation for the task at hand [21]. This can be seen as separating the multiple manifolds that represent data via a series of transformations.

III. Convolutional Neural Networks

Convolutional Neural Networks (CNNs or ConvNets) are probably the most well known Deep Learning model used to solve Computer Vision tasks, in particular image classification. The basic building blocks of CNNs are convolutions, pooling (downsampling) operators, activation functions, and fully-connected layers, which are essentially similar to hidden layers of a Multilayer Perceptron (MLP). Architectures such as AlexNet [6], VGG [22], ResNet [23] and GoogLeNet [24] became very popular, used as subroutines to obtain representations that are then offered as input to other algorithms to solve different tasks.

We can write this network as a composition of a sequence of functions $f_l(.)$ (related to some layer l) that takes as input a vector x_l and a set of parameters W_l, and outputs a vector x_{l+1}:

$$f(x) = f_L\left(\cdots f_2(f_1(x_1, W_1); W_2)\cdots), W_L\right)$$

x_1 is the input image, and in a CNN the most characteristic functions $f_l(.)$ are convolutions. Convolutions and other operators works as building blocks or layers of a CNN: activation function, pooling, normalization and linear combination (produced by the fully connected layer). In the next sections we will describe each one of those building blocks.

A. Convolutional Layer

A layer is composed of a set of filters, each to be applied to the entire input vector. Each filter is nothing but a matrix $k \times k$ of weights (or values) $\mathbf{w}_i$. Each weight is a parameter of the model to be learned. We refer to [18] for an introduction about convolution and image filters.

Each filter will produce what can be seen as an affine transformation of the input. Another view is that each filter produces a linear combination of all pixel values in a neighbourhood defined by the size of the filter. Each region that the filter processes is called local receptive field: an output value (pixel) is a combination of the input pixels in this local receptive field (see Figure 1). That makes the convolutional layer different from layers of an MLP for example; in a MLP each neuron will produce a single output based on all values from the previous layer, whereas in a convolutional layer, an output value $f(i, x, y)$ is based on

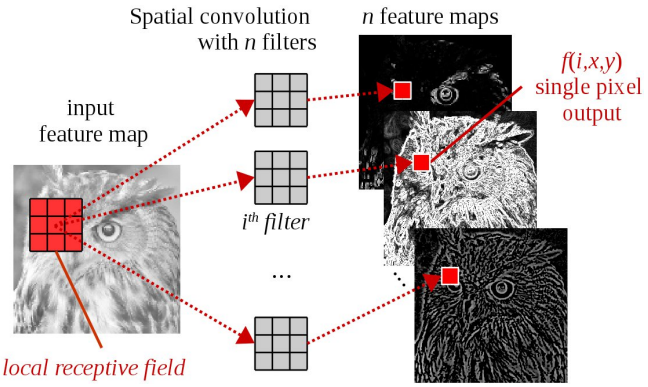


Figure 1. A convolution processes local information centred in each position (x, y): this region is called local receptive field, whose values are used as input by some filter i with weights $\mathbf{w}_i$ in order to produce a single point (pixel) in the output feature map $f(i, x, y)$.

a filter i and local data coming from the previous layer centered at a position (x, y).

For example, if we have an RGB image as an input, and we want the first convolutional layer to have 4 filters of size 5×5 to operate over this image, that means we actually need a $5 \times 5 \times 3$ filter (the last 3 is for each colour channel). Let the input image to have size $64 \times 64 \times 3$, and a convolutional layer with 4 filters, then the output is going to have 4 matrices; stacked, we have a tensor of size $64 \times 64 \times 4$. Here, we assume that we used a zero-padding approach in order to perform the convolution keeping the dimensions of the image.

The most commonly used filter sizes are $5 \times 5 \times d$, $3 \times 3 \times d$ and $1 \times 1 \times d$, where d is the depth of the tensor. Note that in the case of 1×1 the output is a linear combination of all feature maps for a single pixel, not considering the spatial neighbours but only the depth neighbours.

It is important to mention that the convolution operator can have different strides, which defines the step taken between each local filtering. The default is 1, in this case all pixels are considered in the convolution. For example with stride 2, every odd pixel is processed, skipping the others. It is common to use an arbitrary value of stride s, e.g. $s = 4$ in AlexNet [6] and $s = 2$ in DenseNet [25], in order to reduce the running time.

B. Activation Function

In contrast to the use of a sigmoid function such as the logistic or hyperbolic tangent in MLPs, the rectified linear function (ReLU) is often used in CNNs after convolutional layers or fully connected layers [26], but can also be employed before layers in a pre-activation setting [27]. Activation Functions are not useful after Pool layers because such layer only downsamples the input data.

Figure 2 shows plots of those such functions: ReLU cancels out all negative values, and it is linear for all positive

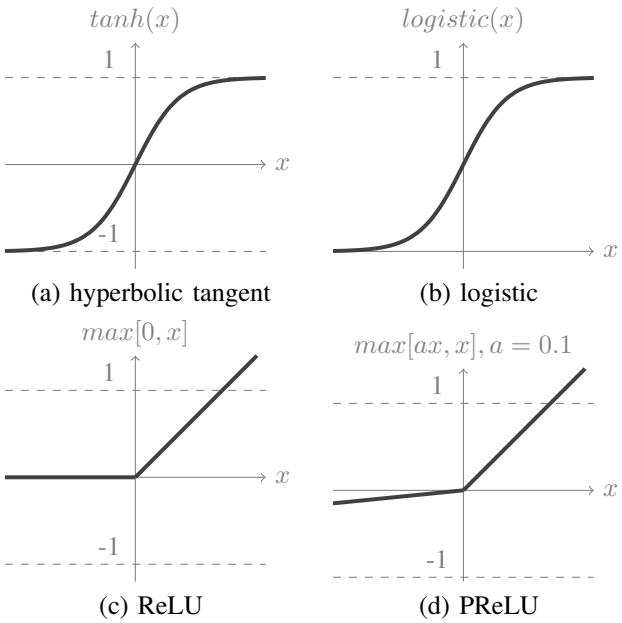


Figure 2. Illustration of activation functions, (a) and (b) are often used in MultiLayer Perceptron Networks, while ReLUs (c) and (d) are more common in CNNs. Note (d) with $a = 0.01$ is equivalent to Leaky ReLU.

values. This is somewhat related to the non-negativity constraint often used to regularize image processing methods based on subspace projections [28]. The Parametric ReLU (PReLU) allows small negative features, parametrized by $0 \leq a \leq 1$ [29]. It is possible to design the layers so that a is learnable during the training stage. When we have a fixed $a = 0.01$ we have the Leaky ReLU.

C. Feature or activation map

Each convolutional neuron produces a new vector that passes through the activation function and it is then called a feature map. Those maps are stacked, forming a tensor that will be offered as input to the next layer.

Note that, because our first convolutional layer outputs a $64 \times 64 \times 4$ tensor, then if we set a second convolutional layer with filters of size 3×3, those will actually be $3 \times 3 \times 4$ filters. Each one independently processes the entire tensor and outputs a single feature map. In Figure 3 we show an illustration of two convolutional layers producing a feature map.

D. Pooling

Often applied after a few convolutional layers, it downsamples the image in order to reduce the spatial dimensionality of the vector. The maxpooling operator is the most frequently used. This type of operation has two main purposes: first, it reduces the size of the data: as we will show in the specific architectures, the depth (3rd dimension) of the tensor often increases and therefore it is convenient to reduce the first two dimensions. Second, by reducing

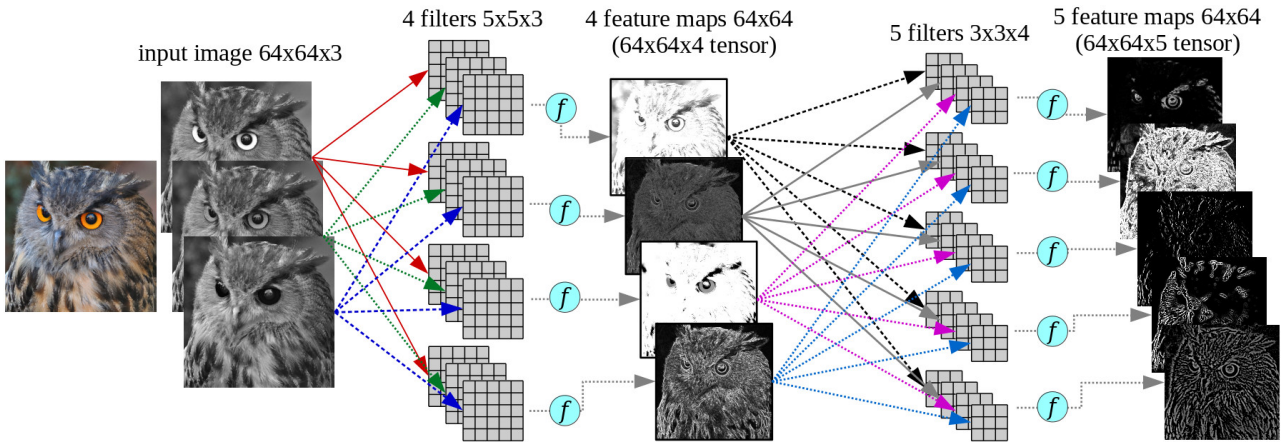


Figure 3. Illustration of two convolutional layers, the first with 4 filters $5 \times 5 \times 3$ that gets as input an RGB image of size $64 \times 64 \times 3$, and produces a tensor of feature maps. A second convolutional layer with 5 filters $3 \times 3 \times 4$ gets as input the tensor from the previous layer of size $64 \times 64 \times 4$ and produces a new $64 \times 64 \times 5$ tensor of feature maps. The circle after each filter denotes an activation function, e.g. ReLU.

the image size it is possible to obtain a kind of multi-resolution filter bank, by processing the input in different scale spaces. However, there are studies in favour to discard the pooling layers, reducing the size of the representations via a larger stride in the convolutional layers [30]. Also, because generative models (e.g. variational AEs, GANs, see Sections IV and V) shown to be harder to train with pooling layers, there is probably a tendency for future architectures to avoid pooling layers.

E. Normalization

It is common also to apply normalization to both the input data and after convolutional layers. In input data preprocessing it is common to apply a z-score normalization (centring by subtracting the mean and normalizing the standard deviation to unity) as described by [31], which can be seen as a whitening process. For layer normalization there are different approaches such as the channel-wise layer normalization, that normalizes the vector at each spatial location in the input map, either within the same feature map or across consecutive channels/maps, using L1-norm, L2-norm or variations.

In AlexNet architecture (2012) [6] the Local Response Normalization (LRN) is used: for every particular input pixel (x, y) for a given filter i, we have a single output pixel $f_i(x, y)$, which is normalized using values from adjacent feature maps j, i.e. $f_j(x, y)$. This procedure incorporates information about outputs of other kernels applied to the same position (x, y).

However, more recent methods such as GoogLeNet [24] and ResNet [23] do not mention the use of LRN. Instead, they use Batch normalization (BN) [32]. We describe BN in Section III-J.

F. Fully Connected Layer

After many convolutional layers, it is common to include fully connected (FC) layers that work in a way similar to a hidden layer of an MLP in order to learn weights to classify the representation. In contrast to a convolutional layer, for which each filter produces a matrix of local activation values, the fully connected layer looks at the full input vector, producing a single scalar value. In order to do that, it takes as input the reshaped version of the data coming from the last layer. For example, if the last layer outputs a $4 \times 4 \times 40$ tensor, we reshape it so that it will be a vector of size $1 \times (4 \times 4 \times 40) = 1 \times 640$. Therefore, each neuron in the FC layer will be associated with 640 weights, producing a linear combination of the vector. In Figure 4 we show an illustration of the transition between a convolutional layer with 5 feature maps with size 2×2 and an FC layer with m neurons, each producing an output based on $f(\mathbf{x}^T\mathbf{w} + b)$, in which $\mathbf{x}$ is the feature map vector, $\mathbf{w}$ are the weights associated with each neuron of the fully connected layer, and b are the bias terms.

The last layer of a CNN is often the one that outputs the class membership probabilities for each class c using logistic regression:

$$P(y = c|\mathbf{x}; \mathbf{w}; b) = \text{softmax}_c(\mathbf{x}^T\mathbf{w} + b) = \frac{e^{\mathbf{x}^T\mathbf{w}_c + b_c}}{\sum_j e^{\mathbf{x}^T\mathbf{w}_j + b_j}},$$

where y is the predicted class, $\mathbf{x}$ is the vector coming from the previous layer, $\mathbf{w}$ and b are respectively the weights and the bias associated with each neuron of the output layer.

G. CNN architecture and its parameters

Typical CNNs are organized using blocks of convolutional layers (Conv) followed by an activation function (AF), eventually pooling (Pool) and then a series of fully connected

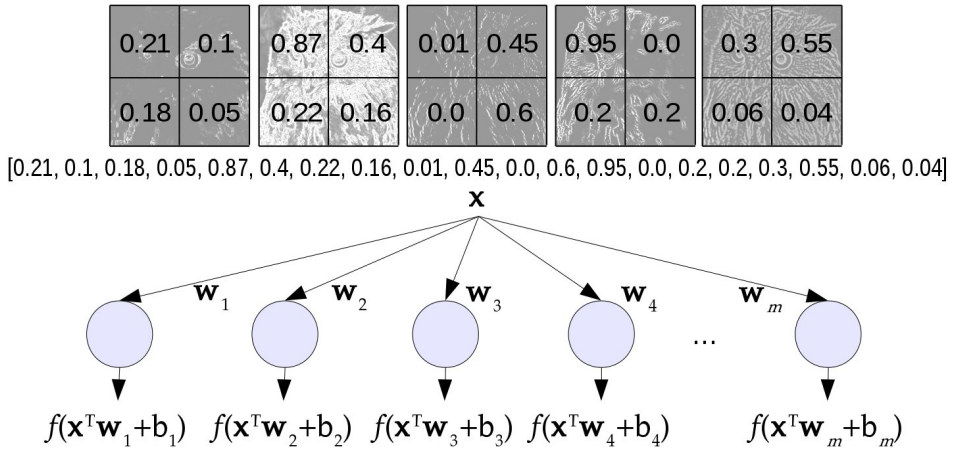


Figure 4. Illustration of a transition between a convolutional and a fully connected layer: the values of all 5 activation/feature maps of size 2×2 are concatenated in a vector $\mathbf{x}$ and neurons in the fully connected layer will have full connections to all values in this previous layer producing a vector multiplication followed by a bias shift and an activation function in the form $f(\mathbf{x}^T\mathbf{w} + b)$.

layers (FC) which are also followed by activation functions. Normalization of data before each layer can also be applied as we describe later in Section III-J.

In order to build a CNN we have to define the architecture, which is given first by the number of convolutional layers (for each layer also the number of filters, the size of the filters and the stride of the convolution). Typically, a sequence of Conv + AF layers is followed by Pooling (for each Pool layer define the window size and stride that will define the downsampling factor). After that, it is common to have a number of fully connected layers (for each FC layer define the number of neurons). Note that pre-activation is also possible as in He *et al*. [27], in which first the data goes through AF and then to a Conv.Layer.

The number of parameters in a CNN is related to the number of weights we have to learn — those are basically the values of all filters in the convolutional layers, all weights of fully connected layers, as well as bias terms.

—*Example*: let us build a CNN architecture to work with RGB input images with dimensions $64 \times 64 \times 3$ in order to classify images into 5 classes. Our CNN will have three convolutional layers, two max pooling layers, and one fully connected layer as follows:

- Conv 1 (Conv $\rightarrow$ AF): 10 neurons $5 \times 5 \times 3$
  - outputs $64 \times 64 \times 10$ tensor
- Max pooling 1: downsampling factor 4
  - outputs $16 \times 16 \times 10$ tensor
- Conv 2 (Conv $\rightarrow$ AF): 20 neurons $3 \times 3 \times 10$
  - outputs $16 \times 16 \times 20$ tensor
- Conv 3 (Conv $\rightarrow$ AF): 40 neurons $1 \times 1 \times 20$
  - outputs $16 \times 16 \times 40$ tensor
- Max pooling 2: downsampling factor 4
  - outputs $4 \times 4 \times 40$ tensor
- FC 1 (FC $\rightarrow$ AF): 32 neurons.
  - outputs 32 values
- FC 2 (output) (FC $\rightarrow$ AF): 5 neurons (one for each class).
  - outputs 5 values

Considering that each of the three convolutional layer's filters has $p \times q \times d$ parameters, plus a bias term, and that the FC layer has weights and bias term associated with each value of the vector received from the previous layer than we have the following number of parameters:

$$
\begin{aligned}
& (10 \times [5 \times 5 \times 3 + 1] = 760) && \text{[Conv1]} \\
+\ & (20 \times [3 \times 3 \times 10 + 1] = 1820) && \text{[Conv2]} \\
+\ & (40 \times [1 \times 1 \times 20 + 1] = 840) && \text{[Conv3]} \\
+\ & (32 \times [640 + 1] = 20512) && \text{[FC1]} \\
+\ & (5 \times [32 + 1] = 165) && \text{[FC2]} \\
=\ & 24097
\end{aligned}
$$

As the reader can notice, even a relatively small architecture can easily have a lot of parameters to be tuned. In the case of a classification problem we presented, we are going to use the labelled instances in order to learn the parameters. But to guide this process we need a way to measure how the current model is performing and then a way to change the parameters so that it performs better than the current one.

H. Loss Function

A loss or cost function is a way to measure how bad the performance of the current model is given the current input and expected output; because it is based on a training set it is, in fact, an empirical loss function. Assuming we want to use the CNN as a regressor in order to discriminate between classes, then a loss $\ell(y, \hat{y})$ will express the penalty for predicting some $\hat{y}$, while the true output value should

be y. The hinge loss or max-margin loss is an example of this type of function that is used in the SVM classifier optimization in its primal form. Let $f_i \equiv f_i(x_j, W)$ be a score function for some class i given an instance x_j and a set o parameters W, then the hinge loss is:

$$\ell_j^{(\text{h})} = \sum_{c \neq y_j} \max\left(0, f_c - f_{y_j} + \Delta\right),$$

where class y_j is the correct class for the instance j. The hyperparameter Δ can be used so that when minimizing this loss, the score of the correct class needs to be larger than the incorrect class scores by Δ at least. There is also a squared version of this loss, called squared hinge loss.

For the softmax classifier, often used in neural networks, the cross-entropy loss is employed. Minimizing the cross-entropy between the estimated class probabilities:

$$\ell_j^{(\text{ce})} = -\log\left(\frac{e^{f_{y_j}}}{\sum_k e^{f_k}}\right), \tag{1}$$

in which $k = 1 \cdots C$ is the index of each neuron for the output layer with C neurons, one per class.

This function takes a vector of real-valued scores to a vector of values between zero and one with unitary sum. There is an interesting probabilistic interpretation of minimizing Equation 1 in which it can be seen as minimizing the Kullback-Leibler divergence between two class distributions in which the true distribution has zero entropy (since it has a single probability 1) [33]. Also, we can interpret it as minimizing the negative log likelihood of the correct class, which is related to the Maximum Likelihood Estimation.

The full loss of some training set (a finite batch of data) is the average of the instances' x_j outputs, $f(x_j; W)$, given the current set of all parameters W:

$$\mathcal{L}(W) = \frac{1}{N}\sum_{j=1}^{N} \ell\left(y_j, f(x_j; W)\right).$$

We now need to minimize $\mathcal{L}(W)$ using some optimization method.

– *Regularization:* there is a possible problem with using only the loss function as presented. This is because there might be many similar W for which the model is able to correctly classify the training set, and this can hamper the process of finding good parameters via minimization of the loss, i.e. make it difficult to converge. In order to avoid ambiguity of solution, it is possible to add a new term that penalizes undesired situations, which is called regularization. The most common regularization is the L2-norm: by adding a sum of squares, we want to discourage individually large weights:

$$R(W) = \sum_k \sum_l W_{k,l}^2$$

We then expand the loss by adding this term, weighted by a hyperparameter λ to control the regularization influence:

$$\mathcal{L}(W) = \frac{1}{N}\sum_{j=1}^{N} \ell\left(y_j, f(x_j; W)\right) + \lambda R(W).$$

The parameter λ controls how large the parameters in W are allowed to grow, and it can be found by cross-validation using the training set.

I. Optimization Algorithms

After defining the loss function, we want to adjust the parameters so that the loss is minimized. The Gradient Descent is the standard algorithm for this task, and the backpropagation method is used to obtain the gradient for the sequence of weights using the chain rule. We assume the reader is familiar with the fundamentals of both Gradient Descent and backpropagation, and focus on the particular case of CNNs.

Note that $\mathcal{L}(W)$ is based on a finite dataset and because of that we are computing Montecarlo estimates (via randomly selected examples) of the real distribution that generates the parameters. Also, recall that CNNs can have a lot of parameters to be optimized, therefore needing to be trained using thousands or millions of images (many current datasets have more than 1TB of data). But if we have millions of examples to be used in the optimization, then the Gradient Descent is not viable, since this algorithm has to compute the gradient for all examples individually. The difficulty here is easy to see because if we try to run an epoch (i.e. a pass through all the data) we would have to load all the examples into a limited memory, which is not possible. Alternatives to overcome this problem are described below, including the SGD, Momentum, AdaGrad, RMSProp and Adam.

– *Stochastic Gradient Descent (SGD):* one possible solution to accelerate the process is to use approximate methods that goes through the data in samples composed of random examples drawn from the original dataset. This is why the method is called Stochastic Gradient Descent: now we are not inspecting all available data at a time, but a sample, which adds uncertainty in the process. We can even compute the Gradient Descent using a single example at a time (a method often used in streams or online learning). However, in practice, it is common to use mini-batches with size B. By performing enough iterations (each iteration will compute the new parameters using the examples in the current mini-batch), we assume it is possible to approximate the Gradient Descent method.

$$W_{t+1} = W_t - \eta \sum_{j=1}^{B} \nabla \mathcal{L}(W; x_j^{\text{B}}),$$

in which η is the learning rate parameter: a large η will produce larger steps in the direction of the gradient, while a small value produces a smaller step in the direction of the

gradient. It is common to set a larger initial value for η, and then exponentially decrease it as a function of the iterations.

In fact, SGD is a rough approximation, producing a non-smooth convergence. Because of that, variants were proposed to compensate for that, such as the Adaptive Gradient (AdaGrad) [34], Adaptive learning rate (AdaDelta) [35] and Adaptive moment estimation (Adam) [36]. Those variants basically use the ideas of momentum and normalization, as we describe below.

– *Momentum:* adds a new variable α to control the change in the parameters W. It creates a momentum that prevents the new parameters W_{t+1} from deviating too much from the previous direction:

$$W_{t+1} = W_t + \alpha(W_t - W_{t-1}) + (1-\alpha)\left[-\eta\nabla\mathcal{L}(W_t)\right],$$

where $\mathcal{L}(W_t)$ is the loss computed using some examples using the current parameters W_t (often a mini-batch). Note that the magnitude of the step for the iteration $t+1$ now is also constrained by the step taken in the iteration t.

– *AdaGrad:* works by putting more weight on rare or infrequent parameters. It creates a history of how much a given parameter already changed the loss, accumulating the individual gradients $g_{t+1} = g_t + \nabla\mathcal{L}(W_t)^2$. Then, the next step is now scaled/normalized for each parameter:

$$W_{t+1} = W_t - \frac{\eta\nabla\mathcal{L}(W_t)^2}{\sqrt{g_{t+1}} + \epsilon},$$

since this historical gradient is computed feature-wise, the infrequent features will have more influence in the next gradient descent step.

– *RMSProp:* computes running averages of recent gradient magnitudes and normalizes using these average so that loosely gradient values are normalized. It is similar to AdaGrad, but here g_t is computed by an exponentially decaying average and not the simple sum of gradients:

$$g_{t+1} = \gamma g_t + (1-\gamma)\nabla\mathcal{L}(W_t)^2$$

g is called the second order moment of $\nabla\mathcal{L}$ (don't confuse it with momentum). The final parameter update is given by adding the momentum:

$$W_{t+1} = W_t + \alpha(W_t - W_{t-1}) + (1-\alpha)\left[-\frac{\eta\nabla\mathcal{L}(W_t)}{\sqrt{g_{t+1}} + \epsilon}\right],$$

– *Adam:* uses an idea that is similar to AdaGrad and RMSProp, but the momentum is used for the first and second order moment so now we have α and γ to control the momentum of respectively W and g. The influence of both decays over time so that the step size decreases when it approaches minimum. We use an auxiliary variable m for clarity:

$$m_{t+1} = \alpha_{t+1} g_t + (1-\alpha_{t+1})\nabla\mathcal{L}(W_t)$$
$$\hat{m}_{t+1} = \frac{m_{t+1}}{1-\alpha_{t+1}}$$

m is called the first order moment of $\nabla\mathcal{L}$ (don't confuse it with momentum) and $\hat{m}$ is m after applying the decaying factor. Then we need to compute the gradients g to use in the normalization:

$$g_{t+1} = \gamma_{t+1} g_t + (1-\gamma_{t+1})\nabla\mathcal{L}(W_t)^2$$
$$\hat{g}_{t+1} = \frac{g_{t+1}}{1-\gamma_{t+1}}$$

g is called the second order moment of $\nabla\mathcal{L}$ (again, don't confuse it with momentum). The final parameter update is given by:

$$W_{t+1} = W_t - \frac{\eta\hat{m}_{t+1}}{\sqrt{\hat{g}_{t+1}} + \epsilon}$$

J. Tricks for Training CNNs

– *Initialization:* random initialization of weights is important to the convergence of the network. The Gaussian distribution $\mathcal{N}(\mu, \sigma)$ is often used to produce the random numbers. However, for models with more than 8 convolutional layers, the use of a fixed standard deviation (e.g. $\sigma = 0.01$) as in [6] was shown to hamper convergence. Therefore, when using rectifiers as activation functions it is recommended to use $\mu = 0$, $\sigma = \sqrt{2/n_l}$, where n_l is the number of connections of a response of a given layer l; as well as initializing all bias parameters to zero [29],.

– *Minibatch size:* due to the need of using SGD optimization and variants, one must define the size of the minibatch of images that is going to be used to train the model taking into account memory constraints but also the behaviour of the optimization algorithm. For example, while a small batch size can make the loss minimization more difficult, the convergence speed of SGD degrades when increasing the minibatch size for convex objective functions [37]. In particular, assuming SGD converges in T iterations, then minibatch SGD with batch size B runs in $O(1/\sqrt{BT} + 1/T)$.

However, increasing B is important to reduce the variance of the SGD updates (by using the average of the loss), and this, in turn, allows you to take bigger step-sizes [38]. Also, larger minibatches are interesting when using GPUs, since it gives better throughput by performing backpropagation with data reuse using matrix multiplication (instead of several vector-matrix multiplications), and needing fewer transfers to the GPU. Therefore, it can be an advantage to choose the batch size so that it fully occupies the GPU memory and choose the largest experimentally found step size. While popular architectures (as we will discuss in Section III-K) use from 32 to 256 examples in the batch size, a recent paper used a linear scaling rule for adjusting learning rates as a function of minibatch size, also adding a warmup scheme with large step-sizes in the first few epochs to avoid optimization problems. By using 256 GPUs and a batch size of 8192, Goyal *et al*. [39] were able to train a Residual Network with 50 layers with the ImageNet in 1 hour.

– *Dropout:* a technique proposed by [40] that, during the forward pass of the network training stage, randomly deactivate neurons of a layer with some probability p (in particular from FC layers). It has relationships with the Bagging ensemble method [41] because, at each iteration of the mini-batch SGD, the dropout procedure creates a randomly different network by subsampling the activations, which is trained using backpropagation. This method became known as a form of regularization that prevents the network to overfit. In the test stage, the dropout is turned off, and the activations are re-scaled by p to compensate those activations that were dropped during the training stage.

– *Batch normalization (BN):* also used as a regularizer, it normalizes the a layer activations at each batch of input data by maintaining the mean activation close to 0 (centering) and the activation standard deviation close to 1, and using parameters γ and β to produce a linear transformation of the normalized vector, i.e.:

$$\mathrm{BN}_{\gamma,\beta}(x_i) = \gamma \left(\frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \right) + \beta, \tag{2}$$

in which γ and β are parameters that can be learned during the backpropagation stage [32]. This allows for example to adjust the normalization, and even restore the data back to its un-normalized form, i.e. when $\gamma = \sqrt{\sigma_B^2}$ and $\beta = \mu_B$.

BN became a standard in the recent years, often replacing the use of both regularization and dropout (e.g. ResNet [23] and Inception V3 [42] and V4).

– *Data-augmentation:* as mentioned previously, CNNs often have a large set of parameters to be optimized, requiring a huge number of training examples. Because it is very hard to have a dataset with sufficient examples, it is common to employ some type of data augmentation. Also, usually images in the same dataset often have similar illumination conditions, a low variance of rotation, pose, etc. Therefore, one can augment the training dataset by many operations in order to produce 5 to 10 times more examples [43] such as: (i) cropping images in different positions — note that CNNs often have a low-resolution image input (e.g. 224×224) so we can find several cropped versions of an image with higher resolution; (ii) flipping images horizontally — and also vertically if it makes sense, e.g. in case of remote sensing and astronomical images; (iii) adding noise [44]; (iv) creating new images by using PCA as in the Fancy PCA proposed by [6]. Note that augmentation must be performed preserving the labels.

– *Pre-processing:* the input data can be pre-processed in several ways: (i) compute the average image for the whole training data and subtracting it from each image; (ii) z-score normalization (as mentioned in Normalization), (iii) PCA whitening that first tries to decorrelate the data by projecting zero-centered original data into the eigenbasis, and then takes the data in the eigenbasis and divides every dimension by the eigenvalue to normalize the scale.

– *Fine-tuning:* when you have a small dataset it can be a challenge to train a CNN. Even data augmentation can be insufficient since augmentation will create perturbed versions of the same images. In this case, it is very useful to use a model already trained in a large dataset (for example the ImageNet dataset [1]), with initial weights that were already learned. To obtain a trained model, adapt its architecture to your dataset and re-enter training phase using such dataset is a process called fine-tuning. Because it often involves the use of a different dataset we discuss this in more detail in Section III-L about Transfer Learning and Feature Extraction.

K. CNN Architectures for Image Classification

There are many proposed CNN architectures for image classification. We chose to cover those that contain significant differences starting with AlexNet, all designed for image classification (1000 classes) of ImageNet Challenge [1]. We refer also to Fukushima's Neocognitron [3] and LeNet [4], both important studies for the history of Deep Learning in Computer Vision. We first describe each architecture and later we show an overall comparison in Table I.

– *AlexNet [6]:* was the champion model in ImageNet Challenge 2012. With ~ 60 million parameters and 650000 neurons, **AlexNet** was originally designed in two branches allowing parallel processing. It uses Local Response normalization, maxpooling with overlapping (window size 3, stride 2), a batch size of 128 examples, momentum of 0.9 weight decay of 0.0005. They initialized the weights using Gaussian-distributed random values with fixed $\sigma = 0.01$, and bias to 1 for the 2nd, 4th and 5th convolutional layers, and bias to 0 for the remaining layers. The learning rate was initialized to 0.01 and arbitrarily dividing this learning rate by 10 three times during the training stage. In Figure 5 we show a diagram of the layers and compare it with the VGG-16, the latter is described next.

– *VGG-Net [22]:* this architecture was developed to increase the depth while making all filters with at most 3×3 size. The rationale behind this is that 2 consecutive 3×3 conv. layers will have an **effective receptive field** of 5×5, and 3 of such layers an effective receptive field of 7×7 when combining the feature maps. The authors claim that stacking 3 conv. layers with 3×3 filters instead of using just one with filter size 7×7 has the advantage of incorporating more rectification layers, making the decision function more discriminative. Also, it incorporates 1×1 conv. layers that perform a linear projection of a position (x, y) across all feature maps in a given layer. There are two most commonly used versions of this CNN: **VGG-16** and **VGG-19**, respectively with 16 weight layers and 19 weight layers. In Figure 5 we show a diagram of the layers of VGG-16 and compare it with the AlexNet.

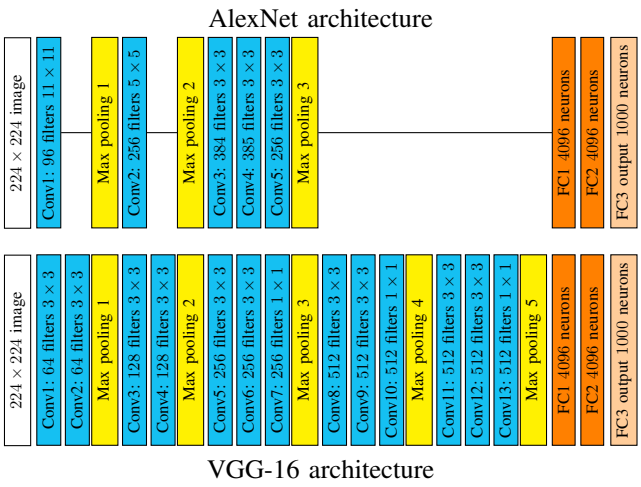


Figure 5. Outline of CNN architectures: AlexNet with variable filter sizes (top), and VGG-16 with fixed 3×3 filter sizes (bottom).

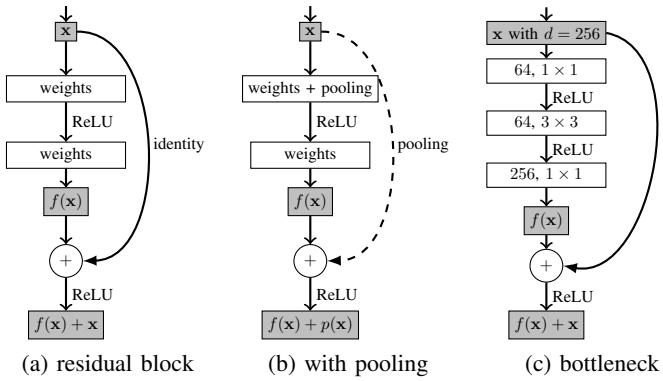


Figure 6. Modules to preserve the characteristics the original vector (identity) allows the vector to skip weight layers, typically skipping 2 layers: (a) the original vector $\mathbf{x}$ before its modification by weights is summed with its transformed version $f(\mathbf{x})$; (b) when some layer also include pooling operation, the dashed line indicates the original vector needed pooling to be summed; (c) in the bottleneck module illustrated, the depth (256) of the input is reduced by the first 1×1 layer of 64 filters and then restored back to 256 by the third layer.

During training, the batch size used is 256. LRN is not used since it shows no classification improvement but increased running time. Maxpooling uses window size 2 and stride 2. Training was regularized by weight decay using L2 regularization $\lambda = 0.0005$, and dropout with 0.5 ratio for the first two FC layers. Learning rate and initialization were similar to those used in AlexNet, but all bias parameters were set to 0 and an initialization followed also a pretraining approach.

– *Residual Networks (ResNet) [23]:* the study describing **ResNets** raises the question of whether we really get better networks by simply stacking more layers. At the time of the publication VGG-19 was considered "very deep", and the authors show that, although depth seems to be correlated with better results, in practice, when increasing the number of layers, the accuracy first saturates and then starts to degrade rapidly and fail to even work in the training set, therefore underfitting.

He *et al.* [23] claim that this could, in fact, be an optimization problem, and propose the use of **residual blocks** for networks with 34 to 152 layers. Those blocks are designed to preserve the characteristics of the original vector $\mathbf{x}$ before its transformation by some layer $f_l(\mathbf{x})$ by skipping weight layers and performing the sum $f_l(\mathbf{x}) + \mathbf{x}$. In Figure 6 we show a diagram of three different versions of such blocks, and in Figure 7 the ResNet architecture with 34 layers that uses the residual block and residual block with pooling. The authors claim that, because the gradient is an additive term, it is unlikely to vanish even with many layers. Interestingly, this architecture does not contain any FC hidden layers. After the last convolution layer, an Average pooling is computed followed by the output layer.

The bottleneck blocks (Figure 6-(c)) are designed to compress the depth of the input feature map via 1×1 convolutions with a reduced number of filters, and then restore its depth by adding another layer of containing a number of filters 1×1 equal to the depth of the input feature map. The bottleneck blocks are used in ResNet with 50, 101 and 152 layers. For example, the ResNet-50 is basically the ResNet-34 replacing all residual blocks (each containing 2 weight layers) with bottleneck blocks (each has 3 weight layers).

For the ImageNet training the authors adopt only Batch Normalization (BN) without regularization, dropout or normalization; data augmentation was performed using crops, horizontal flip and Fancy PCA; they also preprocessed the images by average subtraction. The batch size used was 256 and the remaining parameters are similar to the previously described methods as shown in Table I.

– *GoogLeNet [24] and Inception [42]:* the GoogLeNet as proposed by [24] and the VGGNet [22] achieved similar performances in the 2014 ImageNet challenge [2]. However, the **GoogLeNet** received attention due to its architecture based on modules called **Inception** (see Figure 8). Later improvements in this model are called Inception architectures such as the Inception V3 presented by [42], in which the authors also discuss design principles of CNN architectures including: (i) gentle decrease of representation size from input to output; (ii) use of higher dimensional representations per layer (and consequently more activation maps); (iii) use of lower dimensional embeddings using 1×1 convolutions before spatial convolutions; (iv) balance of width (number of filters per layer) and depth (number of layers).

Recently, the same authors incorporated ideas from ResNets, producing many variants including Inception V4 and Inception-ResNet [45]. Here, for didactic purposes, we

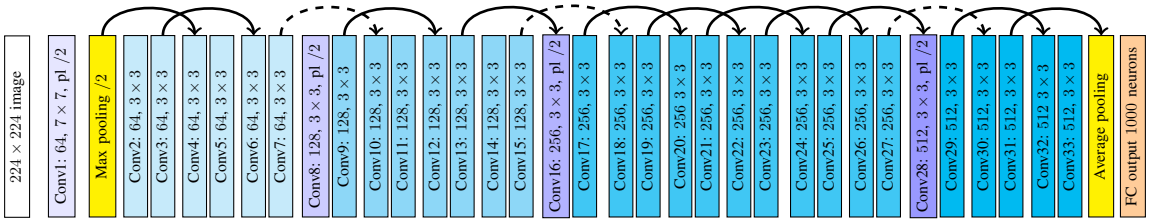
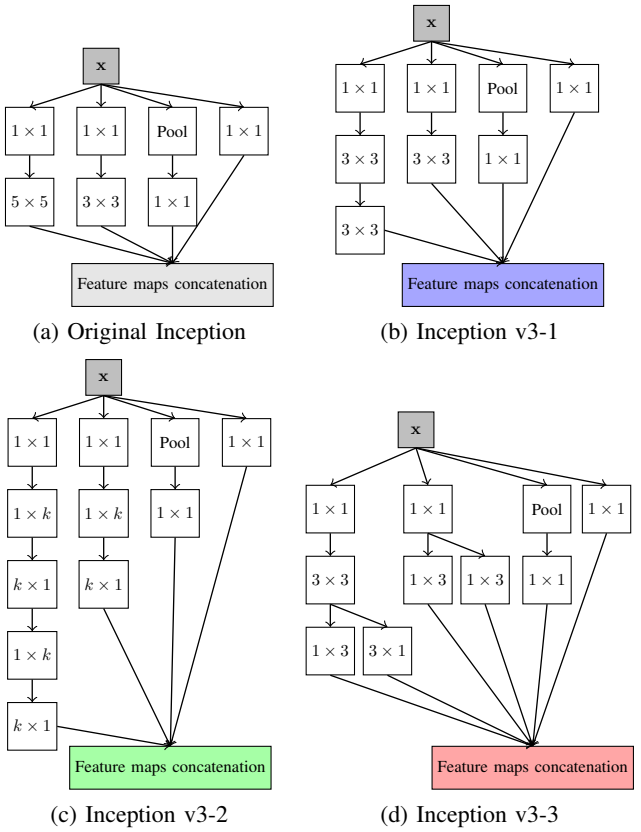


Figure 7. Outline of ResNet-34 architecture. Solid lines indicate identity mappings skipping layers, while dashed lines indicate identity mappings with pooling in order to match the size of the representation in the layer it skips to.



(a) Original Inception (b) Inception v3-1

(c) Inception v3-2 (d) Inception v3-3

Figure 8. Inception modules (a) traditional; (b) replacing the 5×5 by two 3×3 convolutions; (c) with factorization of $k \times k$ convolution filters, (d) with expanded filter bank outputs.

focus on Inception V3 since the V4 is basically a variant of the previous one.

The **Inception module** breaks larger filters (e.g. 5×5, 7×7) that are computationally expensive, into smaller consecutive filters that have the same effective receptive field. Note that this idea was already explored in VGGNet. However, the Inception explores this idea stacking different sequences (in parallel) of small convolutions and concatenates the outputs of the different parallel sequences. The original Inception [24] is shown in Figure 8-(a), while in Inception V3 it follows the design principle (iii) and proposes to factorize a 5×5 convolution into two 3×3 as in Figure 8-(b). In addition, two other modules are proposed: a factorization module for $k \times k$ convolutions as in Figure 8-(c), and an expanded filter bank to increase the dimensionality of representations as shown in Figure 8-(d).

We show the Inception V3 architecture in Figure 9: the first layers include regular convolution layers and pooling, followed by different Inception modules (type V3-1,2,3). We highlight the size of the representation in some transitions of the network. This is important because to show that while the first two dimensions shrink, the third increases following the principles (i), (ii) and (iv). Note that the Inception modules type V3-2 are used with size $k = 7$ for an input representation of spatial size 17×17. The Inception modules type V3-3, that output a concatenation of 6 feature maps per module has the intention of increasing the depth of the feature maps in a coarser scale (spatial size 8×8). Although this architecture has 42 layers (considering internal convolutions layers inside Inception modules), because of the factorization is has a small size in terms of parameters.

For Inception V3 the authors also employ a label-smoothing regularization (LSR) that tries to constrain the last layer to output a non-sparse solution. Given an input example, let k be the index for all classes of a given problem, $Y(k)$ be the ground truth for each class k, $\hat{Y}(k)$ the predicted output and $P(k)$ a prior for each class. The LSR can be seen as using a pair of cross-entropy losses, one that penalizes the incorrect label, and a second that penalizes it to deviate from the prior label distribution, i.e. $(1-\gamma)\ell(Y(k), \hat{Y}(k)) + \gamma\ell(P(k), \hat{Y}(k))$. The parameter γ controls the influence of the LSR and P represent the prior of a given class.

To train the network for ImageNet, they used LSR with a uniform prior, i.e. the same for all classes: $P(k) = 1/1000 \forall k$ for the ImageNet 1000-class problem and $\gamma = 0.1$, BN is applied in both convolutional and FC layers. The optimization algorithm is the RMSProp. All other parameters are standard. They also used gradient clipping with threshold 2.

– *DenseNet [25]:* inspired by ideas from both ResNets and Inception, the **DenseNet** introduces the *DenseBlock*, a sequence of layers where each layer l takes as input

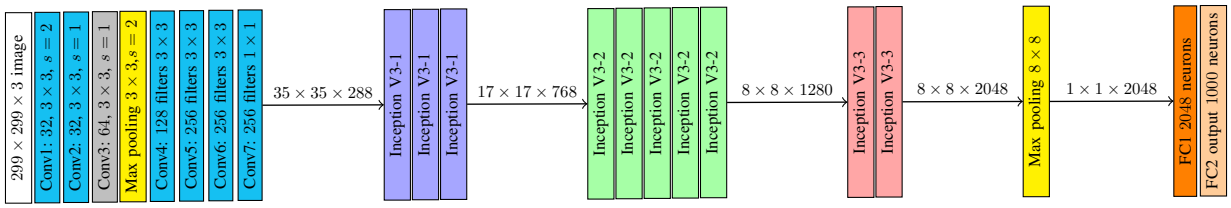


Figure 9. Outline of Inception V3 architecture. Convolutions do not use zero-padding except for Conv3 (in gray). The stride is indicated by s. We highlight the size of the representations before and after the Inception modules types, and also after the last max pooling layer.

all preceding feature maps $\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_{l-1}$ concatenated. Thus, each layer produces an output which is a function of all previous feature maps, i.e.:

$$\mathbf{x}_l = H_l\left([\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_{l-1}]\right).$$

Hence, while regular networks with L layers have L connections, each DenseBlock (illustrated in Figure 10) has a number of connections following an arithmetic progression, i.e. $\frac{L(L+1)}{2}$ direct connections. The DenseNet is a concatenation of multiple inputs of $H_l()$ into a single tensor. Each H_l is composed of three operations, in sequence: batch normalization (BN), followed by ReLU and then a 3×3 convolution. This unusual sequence of operations is called pre-activation unit, was introduced by He *et al*. [27] and has the property of yielding a simplified derivative for each unit that is unlikely to be canceled out, which would in turn improves the convergence process.

Transition layers are layers between DenseBlocks composed of BN, a 1×1 Conv.Layer, followed by a 2×2 average pooling with stride 2.

Variations of DenseBlocks: they also experimented using bottleneck layers, in this case each DenseBlock is a sequence of operations: BN, ReLU, 1×1 Conv., followed by BN, ReLU, 3×3 Conv. This variation is called DenseNet-B. Finally, a compression method is also proposed to reduce the number of feature maps at transition layers with a factor θ. When bottleneck and compression are combined they refer the model as DenseNet-BC.

Each layer has many input feature maps; if each H_l produces k feature maps, then the l^{th} layer has $k_0+k\times(l-1)$ input maps, where k_0 is the number of channels in the input image. However, the authors claim that DenseNet requires fewer filters per layer [25]. The number filters k is a hyperparameter defined in DenseNet as the *growth rate*. For the ImageNet dataset $k = 32$ is used. We show the outline of the DenseNet architecture used for ImageNet in Figure 11.

– *Comparison and other archictectures:* in order to show an overall comparison of all architectures described, we listed the training parameters, size of the model and top1 error in the ImageNet dataset for AlexNet, VGGNet, ResNet, Inception V3 and DenseNet in Table I.

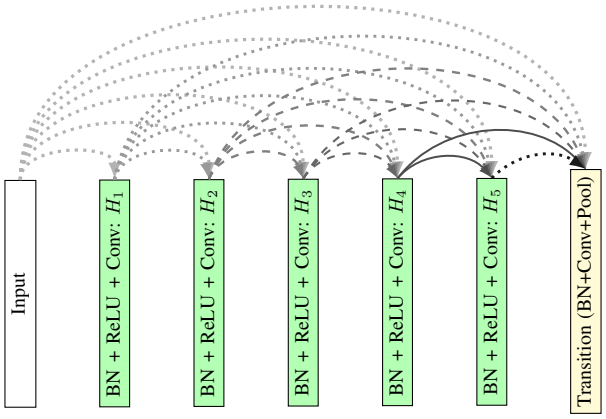


Figure 10. Illustration of a DenseBlock with 5 functions H_l and a Transition Layer.

Finally, we want to refer to another model that was designed to be small, compressing the representations, the **SqueezeNet** [46], based on AlexNet but with $50\times$ fewer parameters and additional compression so that it could, for example, be implemented in embedded systems.

L. Beyond Classification: fine-tuning, feature extraction and transfer learning

When one needs to work with a small dataset of images, it is not viable to train a Deep Network from scratch, with randomly initialized weights. This is due to the high number of parameters and layers, requiring millions of images, plus data augmentation. However, models such as the ones described in this paper (e.g. VGGNet, ResNet, Inception), that were trained with large datasets such as the ImageNet, can be useful even for tasks other than classifying the images from that dataset. This is because the learned weights can be meaningful for other datasets, even from a different domain such as medical imaging [47].

In this context, by starting with some model pre-trained with a large dataset, we can use the networks as **Feature Extractors** [48] or even to achieve **Transfer Learning** [49]. It is also possible to perform **Fine-tuning** of a pre-trained model to create a new classifier with a different set of

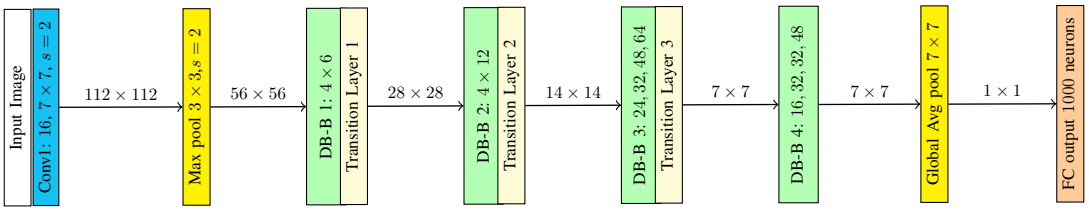


Figure 11. Outline of DenseNet used for ImageNet. The growth rate is $k = 32$, each Conv.layer corresponds to BN-ReLU-Conv(1×1) + BN-ReLU-Conv(3×3). A DenseBlock (DB) 4×6 means there are 4 internal layers with 6 filters each. The arrows indicate the size of the representation between layers.

Table I
COMPARISON OF CNN MODELS IN TERMS OF THE PARAMETERS USED TO TRAIN WITH THE IMAGENET DATASET (INCLUDING AND THEIR ARCHITECTURES

	Training Parameters						Model		ImageNet
CNN	Batch size	Learn.rate	Optm.alg.	Optm.param.	Epochs	Regularization	# Layers	Size	top-1 error
AlexNet	128	0.01	SGD	$\alpha = 0.9$	90	Dropout	8	240MB	37.5%
VGGNet	256	0.01	SGD	$\alpha = 0.9$	74	L2 + Dropout	16-19	574MB (19 lay.)	24.4%
ResNet	256	0.1	SGD	$\alpha = 0.9$	120	BN	34-152	102MB (50 lay.)	19.3%
Inception V3	32	0.045	RMSP	$\alpha, \gamma = 0.9$	100	BN + LSR	42	96MB	18.7%
DenseNet	128	0.1	SGD	$\alpha = 0.9$	100	BN	40-250	30MB (161 lay.)	$\sim$ 18.7%

classes.

In order to perform the aforementioned tasks, there are many options that involve freezing layers (not allowing them to change), discarding layers, including new layers, etc. As we describe next for each case.

– *Building a new CNN classifier via Fine-Tuning:* it is likely that a new dataset has different classes when compared with the dataset originally used to train the model (e.g the 1000-class ImageNet dataset). Thus, if we want to create a CNN classifier for our new dataset based on a pre-trained model, we have to build a new output layer with the number of neurons equal to the number of classes. We then design a training set using examples from our new dataset and use them to adjust the weights of our new output layer.

There are different options involving fine-tuning, for example: (i) allow the weights from all layers to be fine-tuned with the new dataset (not only the last layer), (ii) freeze some of the layers, and allow just a subset of layers to change, (iii) create new layers randomly initialized, replacing the original ones. The strategy will depend on the scenario. Usually, the approach (ii) is the most common, freezing the first layers, but allowing weights of deeper layers to adapt.

– *CNN-based Feature Extractor:* in this scenario, it is possible to make use of pre-trained CNNs even when the dataset is composed of unlabelled (e.g. in clustering problems) or only partially labelled examples (e.g. in anomaly detection problems), in which without label information it is not possible to fine-tuning the weights [50].

In order to extract features a common approach is to perform a forward pass with the images and then use the feature/activation maps of some arbitrary layer as features [48]. For example, using an Inception V3 network, one could get the $1 \times 1 \times 2048$ output from the last Max Pooling layer (see Figure 9), ignoring the output layer. This set of values can be seen as a feature vector with 2048 dimensions to be an input for another classifier, such as the SVM for example. If a more compact representation is needed, one can use dimensionality reduction methods or quantization based on PCA [51] or Product Quantization [52], [53].

– *CNN as a building block:* pre-trained networks can also work as a part of a larger architecture. For example in [54] pre-trained networks for both photo and sketch images are used to compose a triplet network. We discuss those type of architectures in Section VI.

IV. AUTOENCODERS (AEs)

An AE is a neural network that aims to learn to approximate an identity function. In other words, when we feed an AE with a training example x it tries to generate and output $\hat{x}$ that is as similar as possible to x. At first glance, it may seem like such task is trivial: what would be the utility of the AE if it just tries to learn the same data we already have as input? In fact, we are not exactly interested in the output itself, but rather on the representations learned in other layers of the network. This is because the AEs are designed in such way that they are not able to learn a dumb copy function. Instead, they often discover useful intrinsic data properties.

From an architectural point of view, an AE can be divided into two parts: an encoder f and a decoder g. The former takes the original input and creates a restricted (more compact) representation of it – we call this representation **code**. Then, the latter takes this code and tries to reconstruct the original input from it (See Figure 12).

Because the code is a limited data representation, an AE cannot learn how to perfectly copy its input. Instead,

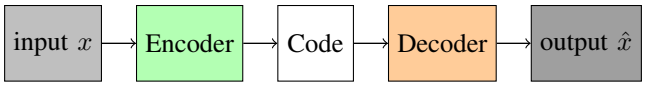


Figure 12. General structure of AEs.

it tries to learn a code that grasps valuable knowledge regarding the structure of the data. In summary, we say that an AE generalizes well when it understands the data-generating distribution – i.e. it has a low reconstruction error for data generated by such mechanism, while having a high reconstruction error for samples that were not produced by it [55].

Convolutional AEs: one can build Convolutional AEs by replacing the fully connected layers of a traditional AE with convolutional layers. Those models are useful because they do not need labelled examples and can be designed to obtain hierarchical feature extraction by making use of the autoencoder architecture. Masci *et al*. [56] described Convolutional Autoencoders for both unsupervised representation learning (e.g. to extract features that can be used in shallow or traditional classifiers) and also to initialize weights of CNNs in a fast and unsupervised way.

Now that we presented the basic structure of an AE and saw the importance of having a limited representation of the data, we give a more in depth explanation concerning how to restrict the generated code. Basically, there are two main ways of achieving this goal: by constructing an undercomplete AE or an overcomplete regularized AE.

A. Undercomplete AEs

In this case the AE has a code that is smaller than its input. Therefore, this code can not hold a complete copy of the input data. To train this model we minimize the following loss function:

$$\mathcal{L}(x, g(f(x))), \tag{3}$$

where L is a loss function (e.g. mean squared error), x is an input sample, g represents the decoder, f represents the decoder, $h = f(x)$ is the code generated by the encoder and $\hat{x} = g(f(x))$ is the reconstructed input data. Let the decoder $f()$ be linear, and $\mathcal{L}()$ be the mean squared error, then the undercomplete AE is able to learn the same subspace as the PCA (Principal Component Analysis), i.e the principal component subspace of the training data [20]. Because of this type of behaviour AEs were often employed for dimensionality reduction.

To illustrate a possible architecture, we show an example of a undercomplete AE in Figure 13, in which we now have the encoder and the decoder composed of two layers each so that the code is computed by $h = f(x) = f_2(f_1(x)))$, and the output by $\hat{x} = g(x) = g_2(g_1(h)))$. Note that, by allowing the functions to be nonlinear, and adding several layers we are increasing the capacity of the AE. In those scenarios, despite the fact that their code is smaller than the input, undercomplete AE still can learn how to copy the input, because they are given too much capacity. As stated in [20], if the encoder and the decoder have enough capacity we could learn a one-dimensional code such that every training sample x^i is mapped to a single neuron in the bottleneck layer using the encoder. Next, the decoder maps this code back to the original input. This particular example – despite not happening in practice – illustrates the problems we may end-up having if an undercomplete AE has too much capacity.

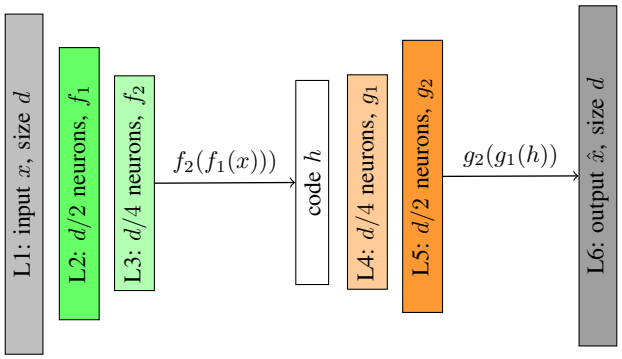


Figure 13. An illustration of a undercomplete AE architecture with: input x, two layers for the encoder which is a sequence of two functions $f_1(f_2(.))$ producing the code h, followed by two layers for the decoder which is a sequence of two functions $g_1(g_2(.))$ producing the output $\hat{x}$.

B. Overcomplete regularized AEs

Differently from undercomplete AEs, in overcomplete AEs the code dimensionality is allowed to be greater than the input size, which means that in principle the networks are able to copy the input without learning useful properties from the training data. To limit their capability of simply copying the input we can either use a *different loss function* via *regularization* (sparse autoencoder) or *add noise to the input data* (denoising autoencoder).

– *Sparse autoencoder:* if we decide to use a different loss function we can regularize an AE by adding a sparsity constraint to the code, $\Omega(h)$ to Equation 3 as follows:

$$\mathcal{L}(x, g(f(x))) + \Omega(f(x)).$$

Thereby, because Ω favours sparse codes, our AE is penalized if it uses all the available neurons to form the code h. Consequently, it would be prevented from just coping the input. This is why those models are known as **sparse autoencoders**. A possible regularization would be an absolute value sparcity penalty such as:

$$\mathcal{L}(x, g(f(x))) + \Omega(h) = \mathcal{L}(x, g(f(x))) + \lambda \sum_i |h_i|,$$

for all i values of the code.

– *Denoising autoencoder:* we can regularize an AE by disturbing its input with some kind of noise. This means that our AE has to reconstruct an input sample x given a corrupted version of it $\tilde{x}$. Then each example is a pair $(x, \tilde{x})$ and now the loss is given by the following equation:

$$\mathcal{L}(x, g(f(\tilde{x}))).$$

By using this approach we restrain our model from copying the input and force it to learn to remove noise and, as a result, to gain valuable insights on the data distribution. AEs that use such strategy are called **denoising autoencoders** (DAEs).

In principle, the DAEs can be seen as MLP networks that are trained to remove noise from input data. However, as in all AEs, learning an output is not the main objective. Therefore, DAEs aim to learn a good internal representation as a side effect of learning a denoised version of the input [57].

– *Contractive autoencoder:* by regularizing the AE using the gradient of the input x, we can learn functions so that the code rate of change follows the rate of change in the input x. This requires a different form for Ω that now depends also on x such as:

$$\Omega(x, h) = \lambda \sum_i \left\| \frac{\partial h}{\partial x} \right\|_F^2,$$

i.e. the squared Frobenius norm of the Jacobian matrix of partial derivatives associated with the encoder function. The **contractive autoencoder**, or CAE, has theoretical connections with denoising AEs and manifold learning. In [58] the authors show that denoising AEs make the reconstruction function to resist small and finite perturbations of x, while contractive autoencoders make the function resist infinitesimal perturbations of x.

The name contractive comes from the fact that the CAE favours the mapping of a neighbourhood of input points (x and its perturbed versions) into a smaller neighbourhood of output points (thus locally contracting), warping the space. We can think of the Jacobian matrix J at a point x as a linear approximation of a nonlinear encoder $f(x)$. A linear operator is said to be contractive if the norm of Jx is kept less than or equal to 1 for all unit-norm of x, i.e. if it shrinks the unit sphere around each point. Therefore the CAE encourages each of the local linear operators to become a contraction.

In addition to the contractive term, we still need to minimize the reconstruction error, which alone would lead to learning the function $f()$ as the identity map. But the contractive penalty will guide the learning process so that we have most derivatives $\frac{\partial f(x)}{\partial x}$ approaching zero, while only a few directions with a large gradient for x that rapidly change the code h. Those are likely the directions approximating the tangent planes of the manifold. These tangent directions ideally correspond to real variations of the data. For instance, if we use images as input, then a CAE should learn tangent vectors corresponding to moving or changing parts of objects (e.g. head and legs) [59].

C. Generative Autoencoders

As mentioned previously, the autoencoders can be used to learn manifolds in a nonparametric fashion. It is possible to associate each of the neurons/nodes of a representation with a tangent plane that spans the directions of variations associated with the difference vectors between the example and its neighbours [60]. An embedding associates each training example with a real-valued vector position [20]. If we have enough examples to cover the curvature and warps of the target manifold, we could, for example, generate new examples via some interpolation of such positions. In this section we focus on generative AEs, but we also describe Generative Adversarial Networks in this paper at Section V.

The idea of using an AE as a generative model is to estimate the density P_{data}, or $P(x)$, that generates the data. For example, in [61] the authors show that when training DAEs including a procedure to corrupt the data x with noise using a conditional distribution $C(\hat{x}|x)$, the DAE is trained to estimate the reverse conditional $P(x|\hat{x})$ as a side-effect. Combining this estimator with the known corruption process, they show that it is possible to obtain an estimator of $P(x)$ through a Markov chain that alternates between sampling from the reconstruction distribution model $P(x|\hat{x})$ (decode), apply the stochastic corruption procedure $C(\hat{x}|x)$ (decode), and iterate.

– *Variational Autoencoders:* in addition to the use of CAEs and DAEs for generating examples, the **Variational Autoencoders** (VAE) emerged as an important method of unsupervised learning of complex distributions and used as generative models [62].

VAEs were introduced as a constrained version of traditional autoencoders that learn an approximation of the true distribution and are currently one of the most used generative models available [63], [64]. Although VAEs share little mathematical basis with CAEs and DAEs, its structure also contains encoder and a decoder modules, thus resembling a traditional autoencoder. In summary, it uses latent variables z that can be seen as a space of rules that enforces a valid example sampled from $P(x)$. In practice, it tries to find a function $Q(z|x)$ (encoder) which can take a value x and give as output a distribution over z values that are likely to produce x.

To illustrate a training-time VAE implemented as a feed-forward network we shown a simple example in Figure 14. In our example $P(x|z)$ is given by a Gaussian distribution, and therefore we have a set of two parameters $\theta = \{\mu, \Sigma\}$. Also, two loss functions are employed: (1) a Kullback-Leibler (KL) divergence $\mathcal{L}_{\text{KL}}$ between the distributions formed by the estimated parameters $\mathcal{N}(\mu(x), \Sigma(x))$ and

$\mathcal{N}(0, I)$ in which I is the identity matrix – note that it is possible to obtain maximum likelihood estimation from minimizing the KL divergence; (2) a squared error loss between the generated example $f(x)$ and the input example x.

Note that this VAE is basically trying to learn the parameters $\theta = \{\mu, \Sigma\}$ of a Gaussian distribution and at the same time a function $f()$, so that, when we sample from the distribution it is possible to use the function $f()$ to generate a new sample. At test time, we just sample z, input it to the decoder and obtain a generated example $f(z)$ (see dashed red line in Figure 14).

For the reader interested in this topic, we refer to [62] for a detailed description of VAEs.

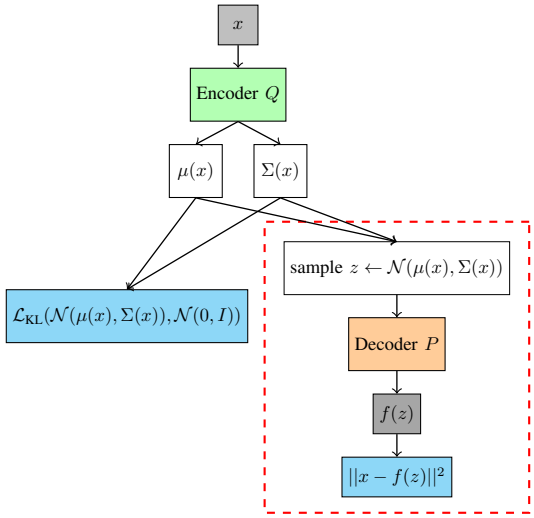


Figure 14. An example of training VAE as a feed-forward network in which we have a Gaussian distributed $P(x|z)$ as in [62]. Blue boxes indicate loss functions. The dashed line highlights the test-time module in which we sample z and generate $f(z)$

V. Generative Adversarial Networks (GAN)

Deep Adversarial Networks have been recently introduced by Goodfellow *et al.* [65] as a framework for building *Generative Models* that are capable of learning, through back-propagation, an implicit probability density function from a set of data samples. One highlight of GANs is that they perform learning and sampling without relying on Markov chains and with scalable performance.

A. Generative Models

Given a training set made of samples drawn from a distribution P_{data}, a generative model learns to produce a probability distribution P_{model} or sample from one such distribution that represents an estimation of its P_{data} counterpart.

One way a generative model can work is by using the principle of *maximum likelihood* (ML). In this context it is assumed that the estimated probability distribution P_{model} is parameterized by a set of parameters θ and that, for any given θ, the likelihood $\mathcal{L}$ is the joint probability of all samples from the training data x "happening" on the P_{model} distribution:

$$\mathcal{L}(x, \theta) = \prod_{i=1}^{m} p_{\text{model}}(x_i; \theta)$$

Training a model to achieve maximum likelihood is then one way of estimating the distribution P_{data} that generated the training samples x. This has similarities with training a VAE via minimization of a Kullback-Leibler divergence (which is a way to achieve ML), as shown in Section IV-C. GANs by default are not based on the ML principle but can, for the sake of comparison, be altered to do so.

Generative models based on this principle can be divided into two main groups: *(a)* models that learn an explicit density function and *(b)* models that can draw samples from an implicit but not directly specified density function. GANs fall into the second group.

There is, however, a plethora of methods from the first group that are currently used in the literature and, in many cases, have incorporated deep learning into their respective frameworks. Fully Visible Belief Networks were introduced by Frey *et al.* [66], [67] and have recently featured as the basis of a WaveNet [68], a state of the art generative model for raw audio. Boltzmann Machines [69] and their counterparts Restricted Boltzmann Machines (RBM) are a family of generative models that rely on Markov Chains for P_{data} approximation and were a key component at demonstrating the potential of deep learning for modern applications [14], [70]. Finally, autoencoders can also be used for generative purposes, in particular denoise, contractive and variational autoencoders as we described previously in Section IV-C.

GANs are currently applied not only for generation of images but also in other contexts, for example image-to-image translation and visual style transfer [71] [72].

B. Generator/Discriminator

In the GAN framework, two models are trained simultaneously; one is a **generator** model that given an input z produces a sample x from an implicit probability distribution P_g; the other is the **discriminator** model, a classifier that, given an input x, produces single scalar (a label) that determines whether x was produced from P_{data} or from P_g.

It is possible to think of the generator as a money counterfeiter, trained to fool the discriminator which can be thought as the police force. The police are then always improving its counterfeiting detection techniques at the same time as counterfeiters are improving their ability of producing better counterfeits.

Formally, GANs are a structured probabilistic model with latent variables z and observed variables x. The two models

are represented generally by functions with the only requirement being that both must be differentiable. The generator is a function G that takes z as an input and uses a list of parameters $\theta^{(G)}$ to output so called observed variables x. The discriminator is, similarly, a function D that takes x as an input and uses a list of parameters $\theta^{(D)}$ to output a single scalar value, the label. The function D is optimized (trained) to assign the correct labels to both training data and data produced by G while G itself is trained to minimize correct assignments of D when regarding data produced by G. There have been many developments in the literature regarding the choice of cost functions for training both G and D. The general framework can be seen as a diagram at Figure 15.

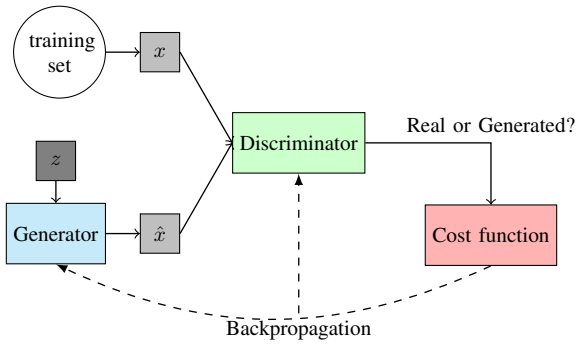


Figure 15. Generative Adversarial Networks Framework.

Given that the only requirement for functions G and D is that both must be differentiable, it is easy to see how MLPs and CNNs fit the place of both.

C. GAN Training

Training GANs is a relatively straightforward process. At each minibatch, values are sampled from both the training set and the set of random latent variables z and after one step through the networks, one of the many gradient-based optimization methods discussed in III-I is applied to update G and D based on loss functions $\mathcal{L}^{(G)}$ and $\mathcal{L}^{(D)}$.

D. Loss Functions

The original proposal [65] is that D and G play a two-player **minimax game** with the value function $V(G, D)$:

$$\min_G \max_D V(G,D) = \mathbb{E}_{x\sim p_{\text{data}}(x)}[\log D(x)] + \mathbb{E}_{z\sim p_g(z)}[\log(1 - D(G(z)))]$$

Which in turn leads to the following loss functions:

$$\mathcal{L}^{(D)}(\theta^{(D)}, \theta^{(G)}) = -\frac{1}{2}\mathbb{E}_{x\sim p_{\text{data}}(x)}[\log D(x)] - \frac{1}{2}\mathbb{E}_z[\log(1 - D(G(x)))]$$

and

$$\mathcal{L}^{(G)}(\theta^{(D)}, \theta^{(G)}) = -J^{(D)}$$

This function and derived loss functions are interesting for their theoretical background; Goodfellow *et al*. showed in [65] that learning in this context is reducible to minimizing a Jensen-Shannon divergence between the data and the model distribution and that it converges if both G and D are convex functions updated in function space. Unfortunately, since in practice G and D are represented by Deep Neural Networks (non-convex functions) and updates are made in parameter space, the theoretical guarantee does not apply.

There is then at least one case where the minimax based cost does not perform well and does not converge to equilibrium. On a minimax game, if the discriminator ever achieves a point where it is performing its task successfully, the gradient of $\mathcal{L}^{(D)}$ approaches zero, but, alarmingly, so does the gradient of $\mathcal{L}^{(G)}$, leading to a situation where the generator stops updating. As suggested by Goodfellow *et al*. [73], one approach to solving this problem is to use a variation of the minimax game, a heuristic **non-saturating game**. Instead of defining $\mathcal{L}^{(G)}$ simply as the opposite of $\mathcal{L}^{(D)}$, we can change it to represent the discriminator's correct hits regarding samples generated by G:

$$\mathcal{L}^{(G)}(\theta^{(D)}, \theta^{(G)}) = -\frac{1}{2}\mathbb{E}_z \log D(G(x))$$

In this game, rather than minimizing log-probability of the discriminator being correct like on the minimax game, the generator maximises the log-probability of the discriminator being mistaken, which is intuitively the same idea but without the vanishing-gradient.

VI. Siamese and Triplet Networks

Whilst Deep Learning using CNN has achieved huge successes in classification tasks, end-to-end regression with DL is also an active field. Notable work in this area are studies of Siamese and Triplet CNNs. The ultimate goal of such networks is to project raw signals into a lower dimensional space (aka. embedding) where their compact representations are regressed so that the similar signals are brought together and the dissimilar ones are pushed away. These networks often consist of multiple CNN branches depending on the number of training samples required in the cost function. The CNN branches can have their parameters either shared or not depending on whether the task is intra- or cross-domain learning.

A. Siamese Networks with Contrastive Loss

A typical Siamese network has two identical CNN branches with associated embedding function $f(.)$. The

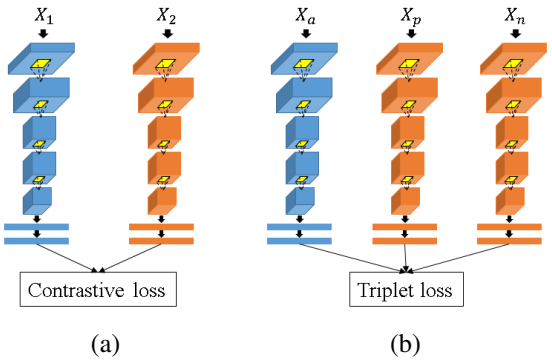


Figure 16. (a) Siamese and (b) triplet networks.

standard cost function for a training exemplar (x_1, x_2) is proposed by Hadsell *et al*. [74]:

$$\mathcal{L}(W,(Y,x_1,x_2)) = \frac{1}{2}(1-Y)(D(W,x_1,x_2))^2 + \frac{1}{2}Y\max\{0, m - D(W,x_1,x_2)\}^2 \quad (4)$$

where $D(W,x_1,x_2) = ||f(W,x_1) - f(W,x_2)||_2$. $Y = 1$ if (x_1,x_2) is a similar pair, and $Y = 1$ otherwise. m is a margin defining a desirable threshold for distance between x_1 and x_2 if they are dissimilar.

Variants of Equation 4 are studied in [75], [76]. Intra-domain regression with shared CNN branches are employed for embedding digits [74], face verification [76], photo retrieval [77] and sketch2edgemap matching [75]. Cross-domain learning between sketch and 3D model was studied in [78] where each CNN branch has its own weights and being updated independently during training.

B. Triplet Networks with Triplet Loss

A triplet network has three branches that accept a similar pair (anchor, positive) and additionally a dissimilar sample serving as the negative example. Unlike the contrastive loss function that enforces a hard regularisation on the similar pair (minimise distance toward zero), the triplet loss regulates a more flexible approach that only consider relative difference between the distances of similar and dissimilar pairs:

$$\mathcal{L}(W,(x_a,x_p,x_n)) = \quad (5)$$

$$\frac{1}{2}\max\{0, m + ||f(W_a,x_a) - g(W_p,x_p)||_2^2 \quad (6)$$

$$- ||f(W_a,x_a) - g(W_p,x_n)||_2^2\} \quad (7)$$

where (x_a,x_p,x_n) is an input triplet, m is the margin and W_a and W_p are the parameters of the anchor $f(.)$ and positive/negative $g(.)$ branches respectively. The positive and negative branches are always identical. The anchor branch is typically shared for learning embedding within a domain

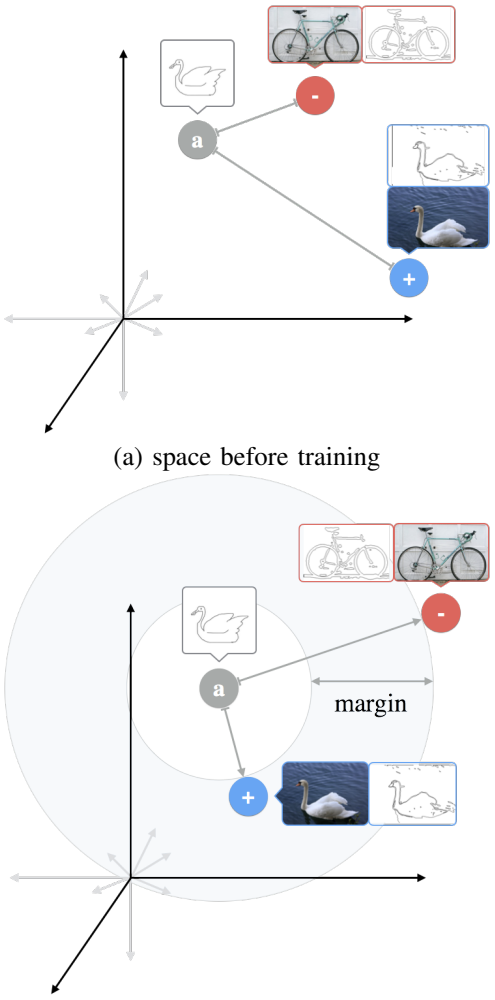


Figure 17. Example of applying a triplet network to a multimodal retrieval problem, the goal in this example is to approximate sketches and natural images of the same category while keeping different categories apart; (a) represents the space as it could be before training and (b) represents the space as it should be after successful training. Triplet loss based learning minimizes the distance between the anchor and positive samples and maximises the distance between the anchor and negative samples.

such as face recognition [79] and tracking [80]. For learning between completely different domains such as sketch vs. photo in sketch-based image retrieval, the anchor branch (fetching sketch) is kept unshared with the image branches [81]. For mapping among similar domains like sketch and image's edgemap, the work in [54] shows that a hybrid half-shared model performs the best. An intuitive visualization of triplet network learning for mapping between sketches and image's edge maps can be seen on Figure 17.

Due to loose regularisation, it is often harder to train a triplet network than a Siamese one. Several variants of Eq. 7 are proposed in [54], [80] to help training triplet networks converged. Furthermore, it is still inconclusive whether or

not triplet CNN is better than Siamese. Contrastive loss is seen to outperform triplet loss in [77], but marginally underperforms its counterpart in [54], [82]. Recently, there exist studies of another loss function that aims to overcome the drawbacks of both contrastive and triplet losses, as claimed in the work of [83].

VII. Applications

Here we describe Computer Vision and Image Processing applications that use the methods detailed in previous sections, by reviewing some of the recent literature on each topic.

A. Visual Stylization

Deep CNNs have also benefited visual content stylization [84]. Echoing the transformation of visual recognition pipelines from hand-engineered to end-to-end trained solutions, deep learning has delivered a step change in our capability to learn artistic styles by example, and to transfer that style to novel imagery - solving a major sub-problem within the Non-Photorealistic Rendering (NPR) field of computer graphics [85].

Stylized output had already been explored in projects such as Google's DeepDream [86], where backpropagation was applied to optimize an input image toward an image maximising a one-hot output on a discriminative network (such as GoogLeNet). Initialising such a network with white noise converges the input toward an optimal trigger image for the desired object category. More interestingly, initialising the optimization from a photograph (plus Gaussian additive i.e. white noise) will hallucinate a locally optimal image in which structures resembling the one-hot object are transformed to more closely represent that object.

Gatys *et al*. [87] went further, observing that such noisy images could optimized to meet a dual objective – seeking a particular content (scene structure and composition) with a particular style (the appearance properties). Specifically, their 'Neural Styles' technique [87] accepted a pair of images (a photograph and an artwork) as input, and sought to hallucinate an input image that matched the content of the input photograph, but matched the style of the input artwork. A simplified structure of the techinque is presented on Figure 18. The input image was again initialised from the photograph plus white noise, and in their work a pre-trained discriminative network (VGG19, trained over ImageNet) was used for the optimization. During optimization network weights are fixed, but the image is converged to minimise loss $\mathcal{L}_{\text{NS}}$:

$$\mathcal{L}_{\text{NS}}(x; p, a) = \alpha \mathcal{L}_{\text{content}}(p, x) + \beta \mathcal{L}_{\text{style}}(a, x)$$

where x refers the image being optimized, p refers to the photograph and a refers to the example style image. Weights $\alpha = 1, \beta = 100$ are typically chosen to balance the two losses, which otherwise exhibit differing orders of magnitude. The content loss compares features extracted from a late (semantic) layer of the VGG19 network — ideally these should match between p and x. The original formulation uses the $conv4\_2$ layer of VGG19 (assume $F(.)$ extracts this image feature):

$$\mathcal{L}_{content} = \frac{1}{2} \sum_{i,j} |F(p) - F(x)|^2$$

The style loss is computed using several Gram (inner-product) matrices, each computed from the feature response from a layer in VGG19. For a given layer l the gram matrix G_l can be computed as an inner product of the vectorised feature map i and j within that layer ($F^l_{i,j}$):

$$G^l_{i,j}(x) = \sum_k F^l_{i,k}(x) F^l_{j,k}(x).$$

The style loss simply sums $\left(G^l(x) - G^l(a)\right)^2$ for several layers l. In some implementations the sum is weighted to afford prominence to certain layers, but this has little practical outcome. The style loss is commonly aggregated over a set of VGG19 layers $\{conv1\_1, conv2\_1, \ldots, conv5\_1\}$.

The use of the Gram matrix to encode style similarity was the core insight in this work, and fundamentally underpins all subsequent work on deep learning for stylization. For example, recent work seeking to stylize video notes that instability in the Gram matrix over time is directly correlated with temporal incoherence (flicker) and seeks to minimize that explicitly in the optimization [88].

Whilst optimization of the input image in this manner yields impressive aesthetics, the technique is slow. More recently, feed-forward networks have been used to stylize images in the order of seconds rather than minutes [89]. Although current feed-forward designs somewhat degrade style fidelity, their interactive speed has enabled video stylization and a slew of commercial (mobile app) software (e.g. Prisma, deepart.io) in the social media space. A discussion of feed-forward architectures for visual stylization is beyond the scope of this tutorial, but we note that contemporary architectures for video stylization integrate two stream networks (again, pre-trained) – one branch dealing with image, the other optical flow, with the latter integrated into the loss function to minimise flicker [88], [90].

B. Image processing (pixels-to-pixels predictions)

Deep Networks for computer vision as seen so far are usually designed for image classification and feature learning. In order to adapt such models to output images (instead of class labels or feature vectors), one must train the networks pixels-to-pixels. This kind of architecture is often called Fully convolutional or Image Processing Deep Networks, from which is possible to obtain for example semantic segmentation [91]–[93], depth estimation [94], [95] and optical flow [96].

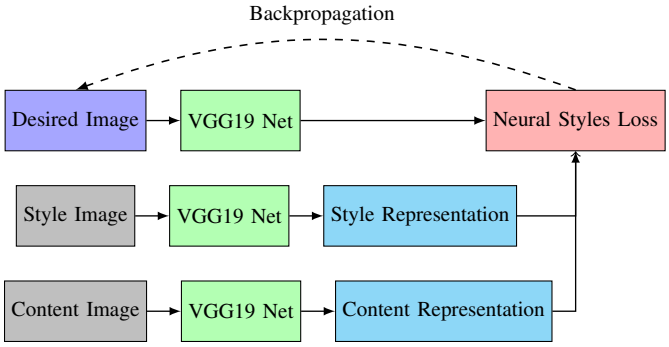


Figure 18. Simplified structure of Gatys *et al*. 'Neural Styles' technique

Fully Convolutional Netwoks (FCNs) are defined as networks that operate on some input of any size and produce an output of corresponding (and possibly resampled) spatial dimensions. FCNs were proposed by [91] for **semantic segmentation**. They show that classification networks can be converted into FCNs that output coarse maps, but in order to provide pixel-wise prediction, those coarse maps must then be connected back to each input pixel. This is performed in [91] via a bilinear interpolation that computes each output from the nearest four inputs by a linear map that depends only on the relative positions of the input and output cells:

$$y_{i,j} = \sum_{\alpha,\beta=0}^{1} |1-\alpha-\{i/f\}||1-\beta-\{i/j\}|x_{\lfloor i/f\rfloor+\alpha,\lfloor j/f\rfloor+\beta},$$

in which f is an upscaling factor, $\{.\}$ returns the fractional part.

Note that upsampling with a factor f can be seen as a *convolution* with fractional input stride of $1/f$. This operation can be used as a upsampling layer (also known as deconvolution layer), and allow end-to-end learning by backpropagation of a pixelwise loss. A stack of deconvolution layers with activation functions can even learn some form of nonlinear upsampling.

An example of FCN is shown in Figure 19, in which the VGGNet-16 is adapted (see Figure 5 for the original model), by replacing FC layers with conv layers and adding deconvolutional (upsampling) layers. To perform upsampling the authors investigated three options: the first, FCN-32s, is a single-stream network that uses just the output of the final layer; the two others explore combinations of outputs from different layers: the FCN-16s combines $2\times$ the output of Conv.15 and $1\times$ the output of MaxPool4, and the FCN-8s combines $4\times$ the output of Conv.15 and $2\times$ the output from MaxPool4 and $1\times$ the output from MaxPool3. The FCN-8s networks obtained the best results for the PASCAL VOC 2011 segmentation dataset.

Another example of image processing network is a CNN-based method that aims to **learn Optical Flow (FlowNet)** [96] that works in a similar way. However, because optical flow needs to be computed over a pair of images (e.g. consecutive frames of a video), two architectures are proposed: the first stacks the pair of images, building an initial input with depth 6 ($2\times$ RGB images) and then using an FCN; the second starts with two parallel streams of convolutions (3 conv.layers), one for each image, and has a fusion layer after the 3rd conv.layer based on a correlation operator.

Depth estimation from stereo image pairs without supervision is proposed by Garg *et al*. [94] using a Convolutional AE. To reconstruct the source image, an inverse warp of the target image is generated using the predicted depth and known inter-view displacement. On the other hand, the problem of estimating depth using a single image still needs supervised learning to achieve good results. In [95], the authors first apply a superpixel algorithm, extracting patches of size 224×224 from the image using the superpixels as centroids. They input individual patches to a CNN composed of 5 conv.layers and 4 FC layers (which they call unary part) and also compute similarities between superpixel neighbours to be feed to one parallel FC layer (they call it pairwise part). Finally, they use a Conditional Random Fields (CRF) loss layer that allows to jointly learn unary and pairwise potentials of a continuous CRF. The pairwise is responsible for enforcing smoothness by minimizing the distance between adjacent superpixels. Because each input of the unary part is a superpixel, it is possible to predict depth for each superpixel patch. They also explored FCNs pretrained on ImageNet with posterior fine-tuning in order to speed-up training time.

C. Video processing and analysis

Video data includes both a spatial domain and a time domain. CNNs architectures as presented in the previous sections are not able to model motion (the temporal dimension of a video). This is because they usually are designed to receive as input an image taken at a given time. In principle it is possible to process a video frame-by-frame, i.e. performing spatial processing only, such as in YOLO method, which achieved static object detection in 155fps [97]. However, the temporal aspect of a video carries crucial information for applications such as activity recognition [98], classification [99], anomaly detection [8], [100], among others. In order to use Deep Learning, we need to input information related to more than one frame so that the spatial-temporal features are learned. Here we describe two possible approaches to address this problem: first with the use of more than one network stream. Note that this is similar to the ideas of Siamese and Triplet networks (see Section VI) and in pixels-to-pixels predictions (Section VII-B). A second approach explores 3D convolutions, as we describe next.

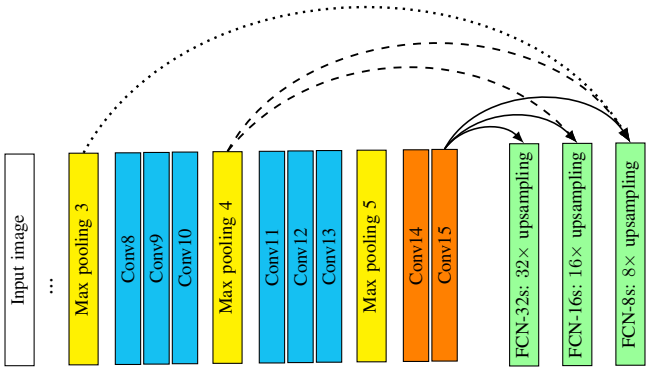


Figure 19. A FCN based on the VGGNet-16 architecture (the layers between input and Max Pooling 3 are omitted). The FC layers 14 and 15 are replaced by convolutional layers. There are 3 versions of deconvolutional/upsampling layers: (FCN-32s) upsample of $32\times$ using the output of Conv15; (FCN-16s) upsample of $16\times$ combining the outputs of Conv15 layer and MaxPool4 layer at stride 16; (FCN-8s) upsample of $8\times$ combining the Conv15 and MaxPool4 layers with an additional MaxPool3 output at stride 8.

– *Multi-stream networks:* deep networks for video use several streams of layers often using images and optical-flow to process video data. The **Two-stream** Network was proposed by [98] and also used in [99]. In this framework, two CNN streams are employed; the first CNN (spatial stream) takes as input frames f_t; the second CNN (temporal stream) takes as input the Optical Flow computed from pairs of L frames. Since each Optical Flow results in a vertical and a horizontal component, the input of the temporal stream CNN is composed of $2L$ input channels. In [98] they report $L = 10$ as giving the best results. Each stream has its own softmax classifier and respective loss function. A fusion of the two class scores is performed (using the sum) to obtain the overall training loss, from which an additional loss function layer is used to compute the backpropagation. In [99], the authors use the Two-Stream network with an LSTM on the top in order to store information over time and explore long-range dynamics.

In the context of anomaly detection, in which we only have examples from the normal class, a three-stream autoencoder was proposed using frames and optical-flow maps to extract appearance, motion and joint representations [8]. Those representations are then used to train a One Class SVM classifier that is used to detect anomalies in video data.

– *Convolutional 3D:* is a convolutional network that considers sequences of frames as input in order to be used for videos. To do so, in [101] the authors propose to use 3D convolutions and 3D pooling layers to create networks that can jointly model the spatial and temporal dimensions of a video. To prove the effectiveness of such approach they developed an architecture, named **Convolutional 3D (C3D)** network.

Similarly to the VGG networks, C3D uses small convolutional kernels ($3 \times 3 \times 3$) to reduce the number of weights to be learned, and 3D pooling layers with sizes $1 \times 2 \times 2$ (Pooling 1), and $2 \times 2 \times 2$ (Pooling 2–5). The network takes as input $224 \times 224 \times 16$ pixels (16 sequential frames, each one with a 224×224 resolution). Note that applying a small 3D convolution in a video volume results in another volume as feature map, which preserves the temporal information to be processed by further layers.

From an architectural point of view, C3D has 8 3D-convolutional, 2 FC, 5 max pooling and one softmax layers. Lastly, it applies a 0.5 dropout on FC6 and FC7. As the authors show, the CD3 learns representations that capture the appearance of the first frames, and thereafter shows activation patterns referring to salient motion. Although the CD3 is able to take as input only fragments of videos containing 16 frames, the method obtained stated-of-the-art results for action recognition in some challenging datasets (e.g. Sports-1M [102] and UCF101 [103])

VIII. Limitations of Deep Learning

Deep learning models for computer vision achieved statistically impressive results. As we show throughout this paper, the idea of stacking layers of operators that are trained to learn hierarchical representations is useful to many applications and contexts. By using different simple architectures such as CNNs, AEs or GANs, and their combination into more complex architectures such as Siamese, Triplet, Multi-stream, etc., it is possible to implement strategies to solve a vast number of applications.

There are, however, clear limitations with this kind of approach. Recall that neural networks (including deep networks) are basically a way to learn a series of transformations applied to an input vector so that it minimizes the loss function. In the case of classification, for example, the network learns how to transform input examples from different classes so that they are mapped to different regions of the output space, one example at a time. This transformation is given by a large set of weights (parameters) that are updated during training stage so that the loss function is minimized. In order to allow training, the transformation must be differentiable, i.e. the map between input and output must be continuous and ideally smooth, which presents a significant constraint [21].

Whilst deep networks need very large annotated datasets, human beings are able to learn a concept by abstraction seeing just a few examples. To train a network to classify numeric digits, we need hundreds or thousands of such digits drawn by different people, while humans usually are able to learn the same concepts with a few examples drawn by one person only. We also easily adapt those concepts over time.

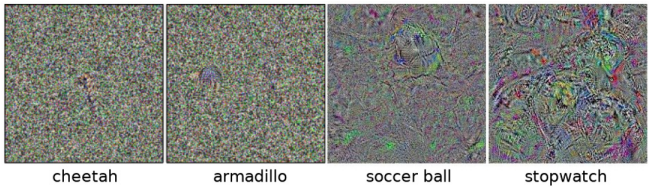


Figure 20. Adversarial examples visually unrecognizable, but classified with more than 99% confidence by Deep Learning methods, as in [104].

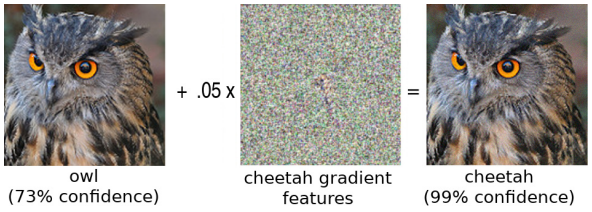


Figure 21. Adversarial image generated by adding a cheetah gradient feature to a picture of an owl, that is then classified with high confidence as a cheetah by a state-of-the-art Deep Network.

Although generative models are a step towards this direction, since they represent an attempt to model the distribution that generates the data categories, those are still a long way from the human learning capacity.

Even considering tasks that are in principle possible to be addressed via Deep Learning, there are many studies showing that perturbations in the input data, such as noise, can significantly impact the results [44]. An interesting paper showed that deep networks can even fail to classify inverted images [105]. It is possible to create images that appear to be just random noise and are unrecognizable to humans, but that CNNs believe to belong to some known class with more than 99% of certainty [104] as examples shown in Figure 20. Based on those unrecognizable patterns, it is possible to design small perturbations (around 4%) added to images so that the output of such methods is completely wrong [106]. In Figure 21 we adapt a famous example by adding a cheetah gradient feature to an owl image so that it is imperceptible to humans, but that is then classified with high confidence as cheetah by a Deep Network. In order to alleviate such fragility to attacks, recent studies recommend to add adversarial examples to the training set or to increase the capacity (in terms of number of filters/nodes per layer), increasing the number of parameters and therefore the complexity of space of functions allowed in the model [107], [108]. Therefore, although the performance of such methods can be impressive when considering overall results, the outputs can be individually unreliable.

If we analyze those already mentioned scenarios in the point of view of Statistical Learning Theory [109], we could hypothesize that those methods are in fact learning the memory-based model. They work well in several cases because they are trained with up to millions of examples, thus unseen data that is similar to training data will eventually fall inside the memorized regions. Although recent work including tensor analysis of representations [110] and uncertainty [111] tried to shed light on learning behaviour of deep networks, more studies on theoretical aspects related to Deep Learning are still needed.

Nevertheless, the positive impact of Deep Learning in Computer Vision and Image Processing is undeniable. By understanding its limitations and inner workings, researchers can explore those methods for many years to come and continue to develop exciting results for new applications.

ACKNOWLEDGMENT

The authors would like to thank FAPESP (grants #2016-16411-4, #2017/10068-2, #2015/04883-0, #2013/07375-0) and the Santander Mobility Award.

REFERENCES

[1] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *CVPR09*, 2009.

[2] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[3] K. Fukushima, "Neocognitron: A hierarchical neural network capable of visual pattern recognition," *Neural networks*, vol. 1, no. 2, pp. 119–130, 1988.

[4] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[5] S. Lazebnik, C. Schmid, and J. Ponce, "Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories," in *Computer Vision and Pattern Recognition, 2006 IEEE Conf. on*, vol. 2. IEEE, 2006, pp. 2169–2178.

[6] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems.*, 2012, pp. 1106–1114.

[7] L. Zhang, L. Zhang, and B. Du, "Deep learning for remote sensing data: A technical tutorial on the state of the art," *IEEE Geoscience and Remote Sensing Magazine*, vol. 4, no. 2, pp. 22–40, 2016.

[8] D. Xu, E. Ricci, Y. Yan, J. Song, and N. Sebe, "Learning deep representations of appearance and motion for anomalous event detection." in *British Matchine Vision Conference (BMVC)*, X. Xie, M. W. Jones, and G. K. L. Tam, Eds. BMVA Press, 2015.

[9] M. Ravanbakhsh, M. Nabi, H. Mousavi, E. Sangineto, and N. Sebe, "Plug-and-play CNN for crowd motion analysis: An application in abnormal event detection," *CoRR*, vol. abs/1610.00307, 2016. [Online]. Available: http://arxiv.org/abs/1610.00307

[10] I. B. Barbosa, M. Cristani, B. Caputo, A. Rognhaugen, and T. Theoharis, "Looking beyond appearances: Synthetic training data for deep cnns in re-identification," *arXiv preprint arXiv:1701.03153*, 2017.

[11] A. Fischer and C. Igel, "Training restricted boltzmann machines: An introduction," *Pattern Recognition*, vol. 47, no. 1, pp. 25–39, 2014.

[12] H. Larochelle, M. Mandel, R. Pascanu, and Y. Bengio, "Learning algorithms for the classification restricted boltzmann machine," *Journal of Machine Learning Research*, vol. 13, no. Mar, pp. 643–669, 2012.

[13] R. Salakhutdinov and G. Hinton, "Deep boltzmann machines," in *Artificial Intelligence and Statistics*, 2009, pp. 448–455.

[14] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural computation*, vol. 18, no. 7, pp. 1527–1554, 2006.

[15] S. Zheng, S. Jayasumana, B. Romera-Paredes, V. Vineet, Z. Su, D. Du, C. Huang, and P. Torr, "Conditional random fields as recurrent neural networks," in *International Conference on Computer Vision (ICCV)*, 2015.

[16] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2013, pp. 6645–6649.

[17] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," in *International Conference on Machine Learning*, 2013, pp. 1310–1318.

[18] R. Gonzalez and R. Woods, *Digital Image Processing*, 3rd ed. Pearson, 2007.

[19] S. Ben-David and S. Shalev-Shwartz, *Understanding Machine Learning: from theory to algorithms*. Cambridge, 2014.

[20] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[21] F. Chollet, *Deep Learning with Python*. Manning, 2017.

[22] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

[23] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015.

[24] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," *CoRR*, vol. abs/1512.00567, 2015.

[25] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[26] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, J. Fürnkranz and T. Joachims, Eds. Omnipress, 2010, pp. 807–814.

[27] K. He, X. Zhang, S. Ren, and J. Sun, *Identity Mappings in Deep Residual Networks*. Cham: Springer International Publishing, 2016, pp. 630–645. [Online]. Available: https://doi.org/10.1007/978-3-319-46493-0_38

[28] M. Ponti, E. S. Helou, P. J. S. Ferreira, and N. D. Mascarenhas, "Image restoration using gradient iteration and constraints for band extrapolation," *IEEE Journal of Selected Topics in Signal Processing*, vol. 10, no. 1, pp. 71–80, 2016.

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

[30] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller, "Striving for simplicity: The all convolutional net," in *ICLR (workshop track)*, 2015. [Online]. Available: http://lmb.informatik.uni-freiburg.de/Publications/2015/DB15a

[31] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, "Efficient backprop," in *Neural networks: Tricks of the trade*. Springer, 2012, pp. 9–48.

[32] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International Conference on Machine Learning*, 2015, pp. 448–456.

[33] M. Ponti, J. Kittler, M. Riva, T. de Campos, and C. Zor, "A decision cognizant Kullback–Leibler divergence," *Pattern Recognition*, vol. 61, pp. 470–478, 2017.

[34] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, no. Jul, pp. 2121–2159, 2011.

[35] M. D. Zeiler, "Adadelta: an adaptive learning rate method," *arXiv preprint arXiv:1212.5701*, 2012.

[36] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations (ICLR)*, 2015.

[37] M. Li, T. Zhang, Y. Chen, and A. J. Smola, "Efficient mini-batch training for stochastic optimization," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2014, pp. 661–670.

[38] L. Bottou, F. E. Curtis, and J. Nocedal, "Optimization methods for large-scale machine learning," *arXiv preprint arXiv:1606.04838*, 2016.

[39] P. Goyal, P. Dollar, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch SGD: Training imagenet in 1 hour." [Online]. Available: https://research.fb.com/publications/imagenet1kin1h/

[40] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *arXiv preprint arXiv:1207.0580*, 2012.

[41] D. Warde-Farley, I. J. Goodfellow, A. Courville, and Y. Bengio, "An empirical analysis of dropout in piecewise linear networks," in *ICLR 2014*, 2014. [Online]. Available: arXivpreprintarXiv:1312.6197

[42] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2818–2826.

[43] K. Chatfield, K. Simonyan, A. Vedaldi, and A. Zisserman, "Return of the devil in the details: Delving deep into convolutional nets," in *British Machine Vision Conference (BMVC)*, 2014, p. 1405.3531.

[44] T. Nazare, G. Paranhos da Costa, W. Contato, and M. Ponti, "Deep convolutional neural networks and noisy images," in *Iberoamerican Conference on Pattern Recognition (CIARP)*, 2017.

[45] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning." in *AAAI*, 2017, pp. 4278–4284.

[46] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size," in *ICLR*, 2016.

[47] H.-C. Shin, H. R. Roth, M. Gao, L. Lu, Z. Xu, I. Nogues, J. Yao, D. Mollura, and R. M. Summers, "Deep convolutional neural networks for computer-aided detection: Cnn architectures, dataset characteristics and transfer learning," *IEEE transactions on medical imaging*, vol. 35, no. 5, pp. 1285–1298, 2016.

[48] A. S. Razavian, H. Azizpour, J. Sullivan, and S. Carlsson, "Cnn features off-the-shelf: An astounding baseline for recognition," in *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops*, ser. CVPRW '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 512–519.

[49] M. Oquab, L. Bottou, I. Laptev, and J. Sivic, "Learning and transferring mid-level image representations using convolutional neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 1717–1724.

[50] S. M. Erfani, S. Rajasegarar, S. Karunasekera, and C. Leckie, "High-dimensional and large-scale anomaly detection using a linear one-class svm with deep learning," *Pattern Recognition*, vol. 58, pp. 121–134, 2016.

[51] M. Ponti, T. S. Nazaré, and G. S. Thumé, "Image quantization as a dimensionality reduction procedure in color and texture feature extraction," *Neurocomputing*, vol. 173, pp. 385–396, 2016.

[52] Y. Kalantidis and Y. Avrithis, "Locally optimized product quantization for approximate nearest neighbor search," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2014, pp. 2321–2328.

[53] T. Bui, L. Ribeiro, M. Ponti, and J. Collomosse, "Compact descriptors for sketch-based image retrieval using a triplet loss convolutional neural network," *Computer Vision and Image Understanding*, 2017.

[54] ——, "Generalisation and sharing in triplet convnets for sketch based visual search," *arXiv preprint arXiv:1611.05301*, 2016.

[55] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 35, no. 8, pp. 1798–1828, Aug. 2013.

[56] J. Masci, U. Meier, D. Cireşan, and J. Schmidhuber, "Stacked convolutional auto-encoders for hierarchical feature extraction," *Artificial Neural Networks and Machine Learning–ICANN 2011*, pp. 52–59, 2011.

[57] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P.-A. Manzagol, "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion," *Journal of Machine Learning Research*, vol. 11, no. Dec, pp. 3371–3408, 2010.

[58] G. Alain and Y. Bengio, "What regularized auto-encoders learn from the data-generating distribution," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 3563–3593, 2014.

[59] S. Rifai, G. Mesnil, P. Vincent, X. Muller, Y. Bengio, Y. Dauphin, and X. Glorot, "Higher order contractive auto-encoder," *Machine Learning and Knowledge Discovery in Databases*, pp. 645–660, 2011.

[60] B. Schölkopf, A. Smola, and K.-R. Müller, "Nonlinear component analysis as a kernel eigenvalue problem," *Neural computation*, vol. 10, no. 5, pp. 1299–1319, 1998.

[61] Y. Bengio, L. Yao, G. Alain, and P. Vincent, "Generalized denoising auto-encoders as generative models," in *Advances in Neural Information Processing Systems*, 2013, pp. 899–907.

[62] C. Doersch, "Tutorial on variational autoencoders," *arXiv preprint arXiv:1606.05908*, 2016.

[63] D. J. Rezende, S. Mohamed, and D. Wierstra, "Stochastic backpropagation and approximate inference in deep generative models," in *Proceedings of the 31st International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, E. P. Xing and T. Jebara, Eds., vol. 32, no. 2. Bejing, China: PMLR, 22–24 Jun 2014, pp. 1278–1286. [Online]. Available: http://proceedings.mlr.press/v32/rezende14.html

[64] D. P. Kingma, "Fast Gradient-Based Inference with Continuous Latent Variable Models in Auxiliary Form," jun 2013. [Online]. Available: http://arxiv.org/abs/1306.0733

[65] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in Neural Information Processing Systems 27*, 2014.

[66] B. J. Frey, G. E. Hinton, and P. Dayan, "Does the wake-sleep algorithm produce good density estimators?" in *Advances in Neural Information Processing Systems 8*, D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, Eds. MIT Press, 1996, pp. 661–667.

[67] B. J. Frey, *Graphical models for machine learning and digital communication*. MIT Press, 1998. [Online]. Available: https://mitpress.mit.edu/books/graphical-models-machine-learning-and-digital-communication

[68] A. van den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "WaveNet: A Generative Model for Raw Audio," sep 2016. [Online]. Available: http://arxiv.org/abs/1609.03499

[69] S. E. Fahlman, G. E. Hinton, and T. J. Sejnowski, "Massively parallel architectures for ai: Netl, thistle, and boltzmann machines," in *Proceedings of the Third AAAI Conference on Artificial Intelligence*, ser. AAAI'83. AAAI Press, 1983, pp. 109–113. [Online]. Available: http://dl.acm.org/citation.cfm?id=2886844.2886868

[70] G. E. Hinton, "Learning multiple layers of representation," pp. 428–434, 2007. [Online]. Available: http://www.cs.toronto.edu/{~}fritz/absps/tics.pdf

[71] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros, "Unpaired image-to-image translation using cycle-consistent adversarial networks," in *ICLR 2017*, 2017, p. arXiv preprint arXiv:1703.10593.

[72] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros, "Image-to-image translation with conditional adversarial networks," in *CVPR 2017*, 2017, p. arXiv preprint arXiv:1611.07004.

[73] I. Goodfellow, "NIPS 2016 Tutorial: Generative Adversarial Networks," dec 2016. [Online]. Available: http://arxiv.org/abs/1701.00160

[74] R. Hadsell, S. Chopra, and Y. LeCun, "Dimensionality reduction by learning an invariant mapping," in *Computer vision and pattern recognition, 2006 IEEE computer society conference on*, vol. 2. IEEE, 2006, pp. 1735–1742.

[75] Y. Qi, Y.-Z. Song, H. Zhang, and J. Liu, "Sketch-based image retrieval via siamese convolutional neural network," in *Image Processing (ICIP), 2016 IEEE International Conference on*. IEEE, 2016, pp. 2460–2464.

[76] S. Chopra, R. Hadsell, and Y. LeCun, "Learning a similarity metric discriminatively, with application to face verification," in *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, vol. 1. IEEE, 2005, pp. 539–546.

[77] F. Radenović, G. Tolias, and O. Chum, "Cnn image retrieval learns from bow: Unsupervised fine-tuning with hard examples," in *European Conference on Computer Vision*. Springer, 2016, pp. 3–20.

[78] F. Wang, L. Kang, and Y. Li, "Sketch-based 3d shape retrieval using convolutional neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 1875–1883.

[79] F. Schroff, D. Kalenichenko, and J. Philbin, "Facenet: A unified embedding for face recognition and clustering," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 815–823.

[80] X. Wang and A. Gupta, "Unsupervised learning of visual representations using videos," in *Proceedings of the IEEE International Conference on Computer Vision*, 2015, pp. 2794–2802.

[81] P. Sangkloy, N. Burnell, C. Ham, and J. Hays, "The sketchy database: learning to retrieve badly drawn bunnies," *ACM Transactions on Graphics (TOG)*, vol. 35, no. 4, p. 119, 2016.

[82] E. Hoffer and N. Ailon, "Deep metric learning using triplet network," in *International Workshop on Similarity-Based Pattern Recognition*. Springer, 2015, pp. 84–92.

[83] O. Rippel, M. Paluri, P. Dollár, and L. D. Bourdev, "Metric learning with adaptive density discrimination," *CoRR*, vol. abs/1511.05939, 2015.

[84] P. Rosin and J. C. (Eds.), *Image and Video based Artistic Stylization*. Springer, 2013.

[85] J.-E. Kyprianidis, J. Collomosse, T. Wang, and T. Isenberg, "State of the 'art': A taxonomy of artistic stylization techniques for images and video," *IEEE Trans. Visualization and Comp. Graphics (TVCG)*, 2012.

[86] A. Mordvintsev, M. Tyka, and C. Olah, "Deepdream," https://github.com/google/deepdream, 2015.

[87] L. Gatys, A. Ecker, and M. Bethge, "A neural algorithm of artistic style," *arXiv preprint arXiv:1508.06576*, 2015.

[88] D. Chen, J. Liao, L. Yuan, N. Yu, and G. Hua, "Coherent online video style transfer," in *Proc. Intl. Conf. Computer Vision (ICCV)*, 2017.

[89] D. Ulyanov, V. Lebedev, A. Vedaldi, and V. Lempitsky, "Texture networks: Feed-forward synthesis of textures and stylized images," in *Proc. Intl. Conf. Machine Learning (ICML)*, 2016.

[90] A. Gupta, J. Johnson, A. Alahi, and L. Fei-Fei, "Stability in neural style transfer: Characterization and application," in *Proc. Intl. Conf. Computer Vision (ICCV)*, 2017.

[91] E. Shelhamer, J. Long, and T. Darrell, "Fully convolutional networks for semantic segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 4, pp. 640–651, 2017.

[92] V. Badrinarayanan, A. Kendall, and R. Cipolla, "Segnet: A deep convolutional encoder-decoder architecture for scene segmentation," *IEEE transactions on pattern analysis and machine intelligence*, 2017.

[93] L.-C. Chen, J. T. Barron, G. Papandreou, K. Murphy, and A. L. Yuille, "Semantic image segmentation with task-specific edge detection using cnns and a discriminatively trained domain transform," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 4545–4554.

[94] R. Garg, G. Carneiro, and I. Reid, "Unsupervised cnn for single view depth estimation: Geometry to the rescue," in *European Conference on Computer Vision*. Springer, 2016, pp. 740–756.

[95] F. Liu, C. Shen, G. Lin, and I. Reid, "Learning depth from single monocular images using deep convolutional neural fields," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 38, no. 10, pp. 2024–2039, 2016.

[96] A. Dosovitskiy, P. Fischer, E. Ilg, P. Hausser, C. Hazirbas, V. Golkov, P. van der Smagt, D. Cremers, and T. Brox, "Flownet: Learning optical flow with convolutional networks," in *Proceedings of the IEEE International Conference on Computer Vision*, 2015, pp. 2758–2766.

[97] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 779–788.

[98] K. Simonyan and A. Zisserman, "Two-stream convolutional networks for action recognition in videos," in *Advances in neural information processing systems*, 2014, pp. 568–576.

[99] Z. Wu, X. Wang, Y.-G. Jiang, H. Ye, and X. Xue, "Modeling spatial-temporal clues in a hybrid deep learning framework for video classification," in *Proceedings of the 23rd ACM international conference on Multimedia*. ACM, 2015, pp. 461–470.

[100] M. Ponti, T. S. Nazare, and J. Kittler, "Optical-flow features empirical mode decomposition for motion anomaly detection," in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2017, pp. 1403–1407.

[101] D. Tran, L. Bourdev, R. Fergus, L. Torresani, and M. Paluri, "Learning spatiotemporal features with 3d convolutional networks," in *Proceedings of the IEEE International Conference on Computer Vision*, 2015, pp. 4489–4497.

[102] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei, "Large-scale video classification with convolutional neural networks," in *CVPR*, 2014.

[103] K. Soomro, A. R. Zamir, and M. Shah, "UCF101: A dataset of 101 human actions classes from videos in the wild," *CoRR*, vol. abs/1212.0402, 2012.

[104] A. Nguyen, J. Yosinski, and J. Clune, "Deep neural networks are easily fooled: High confidence predictions for unrecognizable images," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 427–436.

[105] H. Hosseini and R. Poovendran, "Deep neural networks do not recognize negative images," *arXiv preprint arXiv:1703.06857*, 2017.

[106] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial settings," in *IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2016, pp. 372–387.

[107] A. Rozsa, M. Günther, and T. E. Boult, "Are accuracy and robustness correlated," in *Machine Learning and Applications (ICMLA), 2016 15th IEEE International Conference on*. IEEE, 2016, pp. 227–232.

[108] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," *arXiv preprint arXiv:1706.06083*, 2017.

[109] V. Vapnik, *The nature of statistical learning theory*. Springer science & business media, 2013.

[110] N. Cohen, O. Sharir, and A. Shashua, "On the expressive power of deep learning: A tensor analysis," in *Conference on Learning Theory*, 2016, pp. 698–728.

[111] Y. Gal and Z. Ghahramani, "Dropout as a bayesian approximation: Representing model uncertainty in deep learning," in *International Conference on Machine Learning*, 2016, pp. 1050–1059.