Super Resolution for Automated Target Recognition

Maksim Levental

***Abstract*—Super resolution is the process of producing high-resolution images from low-resolution images while preserving ground truth about the subject matter of the images and potentially inferring more such truth. Algorithms that successfully carry out such a process are broadly useful in all circumstances where high-resolution imagery is either difficult or impossible to obtain. In particular we look towards super resolving images collected using longwave infrared cameras since high resolution sensors for such cameras do not currently exist. We present an exposition of motivations and concepts of super resolution in general and current techniques, with a qualitative comparison of such techniques. Finally we suggest directions for future research.**

1 Artificial Neural Networks

Artificial Neural Network algorithms for SR are all based on a Convolutional Neural Network (CNN) architecture. We first briefly review ANNs in general and CNNs in particular and then explore the recent advances in Deep Learning (DL) for SR.

1.1 Basics

An Artificial Neural Network (ANN or NN) is a function specified by compositions of elementary functions called *artificial neurons*[1] (or simply neurons). Neurons consist of a set of inputs $\mathbf{x} = (x_1, x_2, \dots, x_n)$, an aggregation (typically linear combination), and an activation function σ, which acts as a thresholding mechanism. For example, the simplest function that qualifies as a neuron is a linear function:

$$z(\mathbf{x}) = w_1 x_1 + w_2 x_2 = \sum_i w_i x_i \tag{1}$$

where the the activation function is the trivial one i.e., the identity. Minsky *et al.***minsky2017perceptrons** famously proved that neurons that don't include a non-linear activation function have very little representational power (i.e., they're unable to represent functions as simple as even XOR[2].) and those that do are universal representers (i.e., given enough layers and neurons they're able to represent functions of arbitrary complexity). Common non-trivial, i.e., non-linear, activation functions are the sigmoid function

$$S(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1} \tag{2}$$

or the hyperbolic tangent function

$$\tanh x = \frac{\sinh x}{\cosh x} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x}-1}{e^{2x}+1} \tag{3}$$

or the piece-wise defined *rectified linear unit* (ReLU)

$$\begin{aligned} \mathrm{ReLU}(x) &:= \begin{cases} x & \text{if } x > 0, \\ 0 & \text{otherwise} \end{cases} \qquad (4) \\ &= \max(0, x) \qquad (5) \end{aligned}$$

Note that eqn. (**??**) passes through the origin $(0,0,0)$ since it has no constant term; in the parlance of machine learning the neuron is missing a bias term[3] b:

$$z(\mathbf{x}) = \sum_i w_i x_i + b \tag{6}$$

Neurons can be represented as directed graphs where a vertex represent an input or a neuron and edges represent the weights in the linear combination (see figure **??**). ANNs are then assemblies of neurons grouped into *layers* with the layers composed by applying neurons to outputs from immediately preceding layers. Those layers that are not input or output layers are denoted *hidden* layers. For example the ANN specified in figure **??** represents the function

$$\begin{aligned} y(\mathbf{x}) &= \sigma' \left(\sum_j w'_j z_j(\mathbf{x}) + b' \right) \\ &= \sigma' \left(\sum_{j=1}^{m} w'_j \sigma \left(\sum_{i=1}^{n} w_i x_i + b_j \right) + b' \right) \end{aligned} \tag{7}$$

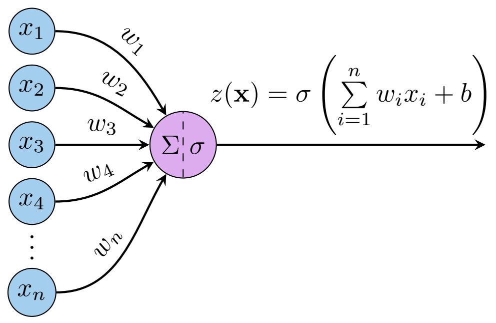
If ANNs were simply another way to diagrammatically represent non-linear functions they would be fairly uninteresting. In fact ANNs are comprised by their definition and a *learning* method. The learning method enables the function to approximate some other function, by adjusting the weights w_i, given *training* pairs $\{\mathbf{x}_k, t_k\}$ where $\mathbf{x}_k$ is the kth training *sample* and t_k is the kth training *target*. The most common such learning rule is called the Delta rule**widrow1960adaptive** for a single neuron, which can be derived from minimizing the the squared error *loss* with respect to each of the weights for a given training pair $(\mathbf{x}_k, t_k)$:

$$L(w_1, \dots, w_n) = \sum_k \frac{1}{2}(t_k - y(\mathbf{x}_k))^2 \tag{8}$$

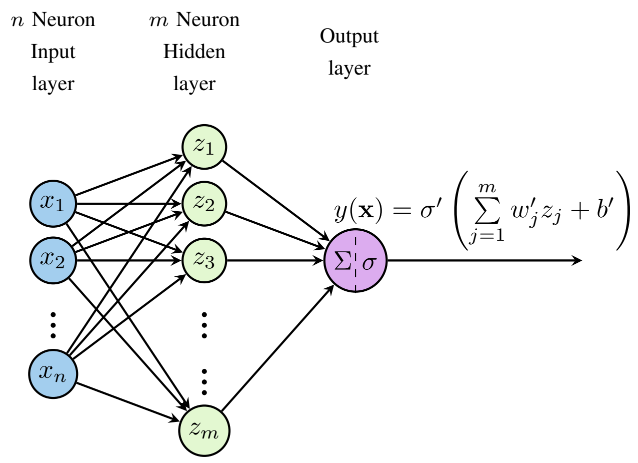
[1]Artificial neurons are loosely inspired by the spiking neuron model of biological neurons, where dendrites correspond to inputs, the soma corresponds to a linear sum of inputs, and the axon corresponds to the activation function (since it propagates an action potential measured in the soma depending on a threshold).

[2]Exclusive-Or $\mathrm{XOR}(a, b)$ is a Boolean function (maps $\{0,1\}$ to $\{0,1\}$) defined to be 1 when $a = b$ and 0 otherwise.

[3]The semantics of bias are inherited from from statistics where the bias of an *estimator* is the extent to which the estimator, on average, differs from the value being estimated. Therefore, if we regard an artificial neuron as an estimator (i.e., a linear regression model), we see that the constant term w_0 precisely determines its bias.



(a) Single Artificial Neuron function.



(b) Multi-layer ANN.

Fig. 1: Artificial Neural Network representation.

and hence

$$\frac{\partial L}{\partial w_i} = -\sum_k (t_k - y(\mathbf{x}_k)) \cdot y' \cdot x_{ik} \tag{9}$$

where here by y' we mean the derivative of the activation function with respect to its argument and by x_{ik} we mean the ith input x_i of the kth training sample. Hence, by gradient descent the weights w_i should be adjusted in the opposite direction of $\frac{\partial L}{\partial w_i}$ and so we have the weight adjustment rule

$$\Delta w_i = \alpha \cdot \sum_k (t_k - y(\mathbf{x}_k)) \cdot \sigma' \cdot x_{ik} \tag{10}$$

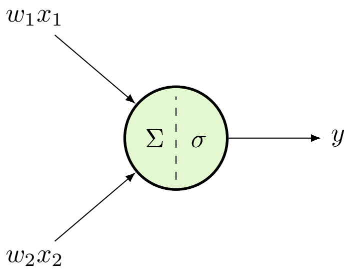
where α is a small constant called the *learning rate*.

The Delta rule is essentially the chain rule as applied to ANNs. In general computing the partial derivatives $\frac{\partial L}{\partial w_i}$ for a deep (many layers) and wide (many neurons in each layer) network is onerous. To mititage the effect of this combinatorial explosion of dependencies between the weights Rumelhart *et al.***rumelhart1988learning** popularlized a technique called *back-propagation*[4] or simply backprop (see figure **??**). Another inefficiency of the Delta rule is that it requires evaluating the ANN on the entire batch of samples in order to compute the adjustment Δw_i. For large training sets (on the order of millions of samples) this is infeasible due to memory limitations. Stochastic Gradient Descent (SGD) replaces computing the *batch loss* $\sum_k (t_k - y(\mathbf{x}_k))$ in eqn. (**??**) with an iterative update to w_i:

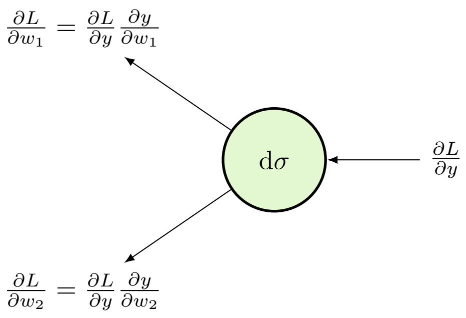
$$w_i^k = w_i^{k-1} + \alpha \cdot (t_k - y(\mathbf{x}_k)) \cdot \sigma' \cdot x_{ik} \tag{11}$$

[4]Backprop itself is a particular instance of a general technique called automatic differentiation (AD). AD computes exact derivatives of functions in either *forward accumulation* or *reverse accumulation***linnainmaa1976taylor**. Backprop corresponds to reverse mode. In reverse mode the dependent variable y to be differentiated is fixed and the derivative is computed with respect to each sub-expression z_i recursively:

$$\frac{\partial y}{\partial w} = \frac{\partial y}{\partial z_1}\frac{\partial z_1}{\partial w} = \left(\frac{\partial y}{\partial z_2}\frac{\partial z_2}{\partial z_1}\right)\frac{\partial z_1}{\partial w} = \left(\left(\frac{\partial y}{\partial z_3}\frac{\partial z_3}{\partial z_2}\right)\frac{\partial z_2}{\partial z_1}\right)\frac{\partial z_1}{\partial w} = \cdots$$



(a) Forward Pass.



$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial w_1}$
$\mathrm{d}\sigma$
$\frac{\partial L}{\partial y}$
$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial w_2}$

(b) Backwards Pass. Note that $\frac{\partial L}{\partial y}$ can be reused when computing both $\frac{\partial L}{\partial w_1}$ and $\frac{\partial L}{\partial w_2}$.

Fig. 2: Back-propagation computation of derivatives.

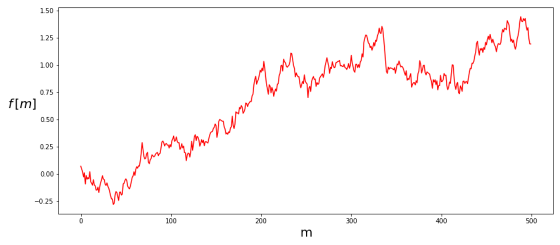
Equation (**??**) is evaluated for each of the k samples sequentially and therefore saves having to store all training samples in memory.

1.2 Convolutional Neural Networks

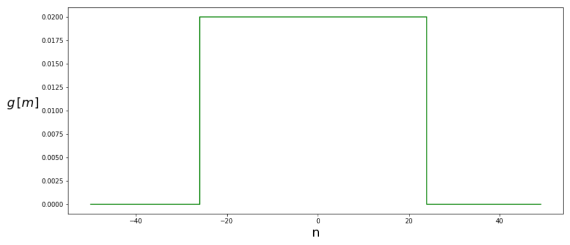
The one-dimensional (1-D) discrete convolution $(f * g)$ of 1-D discrete functions f, g is defined

$$(f * g)[n] := \sum_i f[i]g[n-i] \tag{12}$$

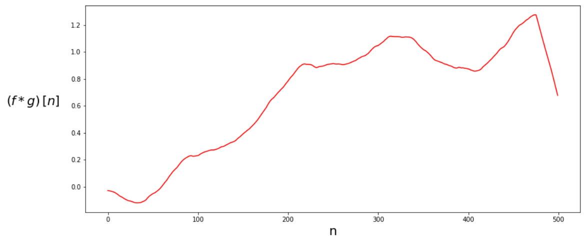
The convolution of a function f with a finite sequence of values g can be interpreted as filtering f with the filter g;



(a) Noisy function f (Wiener process sample).



(b) Heaviside function (low-pass filter g).



(c) Smoothed (low-pass filtered) $f * g$.

Fig. 3: Convolution as filtering.

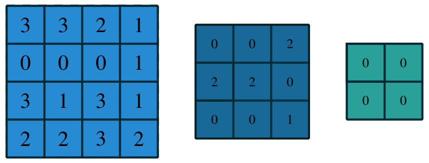
for example convolving a noisy f with a Heaviside function is effectively low-pass filtering f (see figure **??**). The the sequence of values that comprise g is called the *kernel* of g and the length of the sequence is called the *bandwidth* or just the width.

The two-dimensional (2-D) discrete convolution $(f * g)$ of 2-D discrete functions f, g is defined

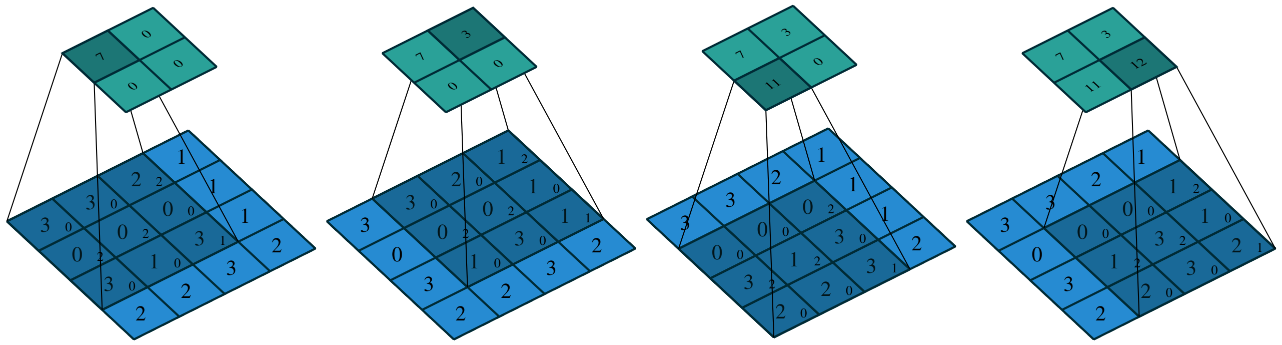
$$(f * g)[n, m] \coloneqq \sum_i \sum_j f[i, j] g[n - i, m - j] \tag{13}$$

and can be interpreted in exactly the same way as for 1-D convolutions. For 2-D convolutions the kernels are most often square and therefore the kernel dimensions are specified rather than just width (see figure **??**).

1.3 Deep Neural Networks

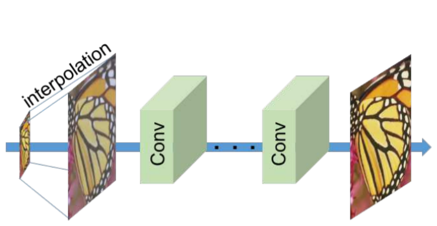


(a) Input f, 2-D 3×3 convolution kernel g, and the output (prior to evaluation of the convolution).

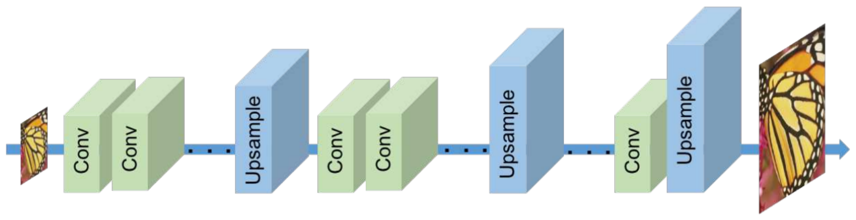


(b) Evaluation of the convolution. More strongly shaded elements in the input show the *receptive field* of each element in the output.**dumoulin2016guide**.

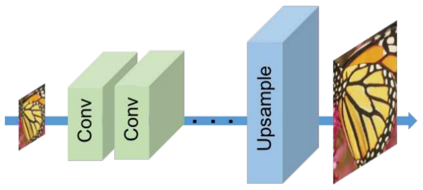
Fig. 4: 2-D convolution.



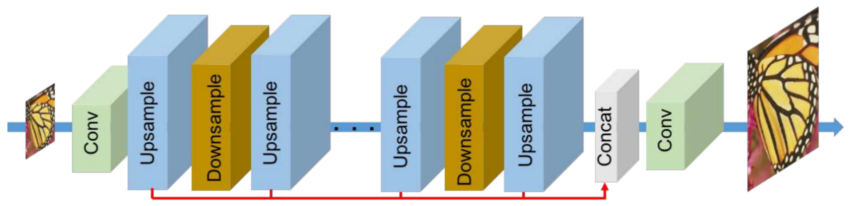
(a) Predefined up-sampling.



(b) Progressive up-sampling.



(c) Single up-sampling.



(d) Iterative up-down-up-sampling.

Fig. 5: Comparison of Deep SR network architectures**haris2018deep**