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 (ANN or NN) algorithms for SR are all based on a Convolutional Neural Network (CNN) architecture. We first briefly review ANNs in general, CNNs in particular, powerful architectures called Deep Neural Networks (DNNs) and techniques that make their use feasible (Deep Learning). We then proceed to review applications of DNNs to SR.

1.1 Basics

An ANN is a function specified by compositions of elementary functions called *artificial neurons*[1] (or simply neurons). Neurons consist of a set of inputs $\mathbf{x} := (x_1, x_2, \ldots, x_n)$, an aggregation (typically linear combination), and an textitactivation 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. Common non-trivial, i.e., non-linear, activation functions are the sigmoid function

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

or the hyperbolic tangent function

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

[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).

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

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

Minsky *et al.***minsky2017perceptrons** famously proved that neurons that don't include a non-linear activation function have very little representational power and those that do are universal representers[2]. 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

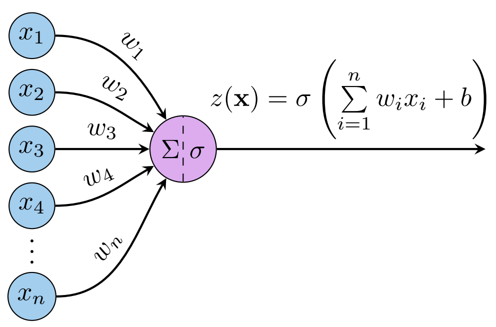
$$\begin{aligned} y(\mathbf{x}) &= \sigma'\left(\sum_j w'_j z_j(\mathbf{x}) + b'\right) \\ &= \sigma'\left(\sum_{j=1}^{m} w'_j \sigma\left(\sum_{i=1}^{n} w_i x_i + b_j\right) + b'\right) \end{aligned} \tag{7}$$

If ANNs were simply another way to diagrammatically represent non-linear functions they would be fairly uninteresting. In fact ANNs entail the definition of a non-linear function 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 $(\mathbf{x}_k, t_k)$:

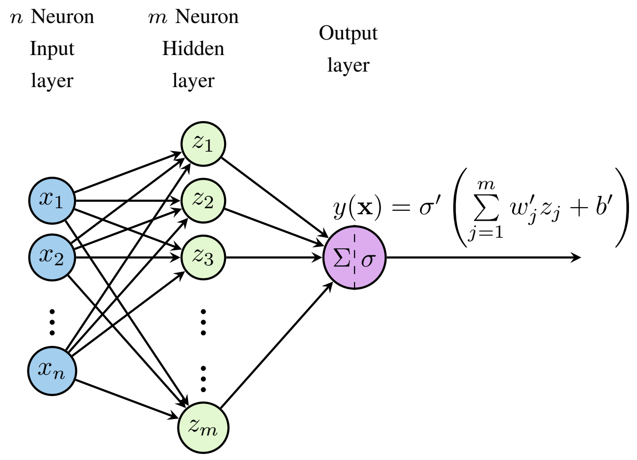
$$L(w_1, \ldots, w_n) := \sum_k \frac{1}{2}(t_k - y(\mathbf{x}_k))^2 \tag{8}$$

[2]Given enough layers and neurons ANNs with non-linear activations are able to represent functions of arbitrary complexity. Conversely, sans a non-linear activation function, ANNs are unable to represent functions as simple as even XOR.

[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

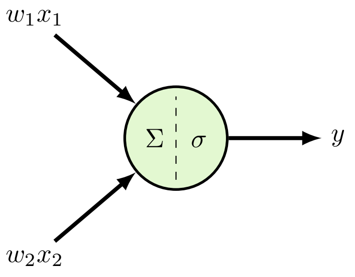
$$\frac{\partial L}{\partial w_i} = -\sum_k (t_k - y(\mathbf{x}_k)) \cdot y' \cdot x_{ik} \tag{9}$$

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

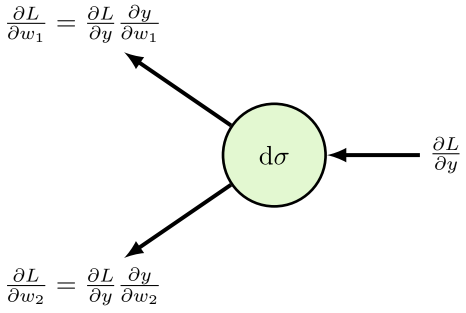
$$\Delta w_i \coloneqq \alpha \cdot \sum_k (t_k - y(\mathbf{x}_k)) \cdot \sigma' \cdot x_{ik} \tag{10}$$

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 set 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



(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.

total $\sum_k (t_k - y(\mathbf{x}_k))$ in eqn. (**??**) with an iterative update to w_i:

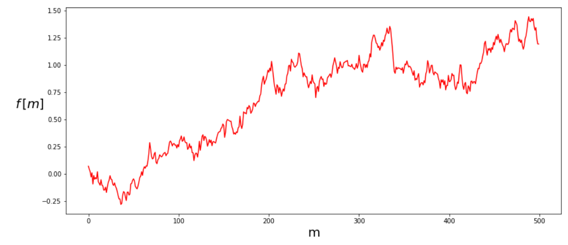
$$\Delta w_i^t = w_i^t - w_i^{t-1} \coloneqq \alpha \cdot \sum_j (t_j - y(\mathbf{x}_j)) \cdot \sigma' \cdot x_{ij} \tag{11}$$

where $\sum_j (t_j - y(\mathbf{x}_j))$ is the loss over a subset of samples called a batch and the aggregate weight update

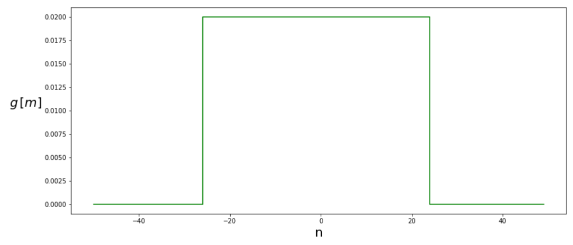
$$\Delta w_i = \sum_t \Delta w_i^t \tag{12}$$

[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:

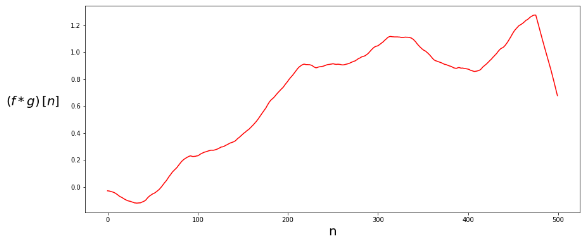
$$\frac{\partial y}{\partial w} = \frac{\partial y}{\partial z_1}\frac{\partial z_1}{\partial w} = \left(\frac{\partial y}{\partial z_2}\frac{\partial z_2}{\partial z_1}\right)\frac{\partial z_1}{\partial w} = \left(\left(\frac{\partial y}{\partial z_3}\frac{\partial z_3}{\partial z_2}\right)\frac{\partial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



(a) 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.

1.2 Convolutional Neural Networks

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

$$(f * g)[n] := \sum_i f[i]g[n-i] \tag{13}$$

The convolution of a function f with a finite sequence of values g can be interpreted as filtering f with the filter g; for example convolving a noisy f with a Heaviside function is effectively low-pass filtering f (see figure **??**). The sequence of values that comprise g is called the *kernel* of the filter g and the length of the sequence is called the *bandwidth* of g (or simply the width). The two-dimensional (2-D) discrete convolution $(f * g)$ of 2-D discrete functions f, g is defined

$$(f * g)[n, m] := \sum_i \sum_j f[i, j]g[n-i, m-j] \tag{14}$$

and can be interpreted in exactly the same way as 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 **??**).

Notice that eqns. (**??**) and (**??**) are completely linear in their inputs $f[i]$ ($f[i, j]$) with weights $w_i = g[n-i]$ ($w_{ij} = g[n-i, m-j]$) and hence naturally constitute a neuron (layer of neurons); a *convolution layer* in a multi-layer ANN is either eqn. (**??**) or eqn. (**??**) with kernel values being iteratively adjusted by the learning rule. Hence, a CNN is an ANN with one or more convolution layers; CNNs are particularly effective for tasks that operate on images (since image patches have more semantic signficance than image slices). In practice multiple filters are applied to the same input and then stacked to produce a higher-dimensional output (see figure **??**), each dimension of which is called a *feature map*. Depending on whether the CNN is being employed to solve a generative task or a classification task the activation function might be either a ReLU (applied element-wise to the output of the convolution layer) or a *max-pooling* filter:

$$\text{max-pool}(f, g)[n, m] := \max_{i,j} \left[f[i, j]g[n-i, m-j] \right] \tag{15}$$

There are many other convolution operators (e.g., strided, dilated, transposed) that are beyond the scope of this survey**dumoulin2016guide**.

1.3 Deep Neural Networks

Deep Neural Networks (DNNs) are ANNs that have multiple layers and many neurons in each layer. Intuitively the advantage of deep networks (over shallow networks) is they learn[5] hierarchies of concepts (called *features*); for example in face recognition tasks, layers proximal to the input layer learn to recognize elementary features such as edges, layers distal to the input layer learn abstractions such as arrangements of features that comprise eyes or noses, and layers even more distal to the input layer learn entire faces.

Training DNNs presents many challenges; due to their depth they suffer from issues such as *vanishing gradients* and *overfitting*. Vanishing gradients is an all but complete cessation of substantive updates to weights; consider the partial derivative of the activation function σ' in eqn. (**??**). Notice that if $|\sigma'| \ll 1$ then Δw_i will be very small. This occurs for a single neuron when the input *saturates* the activation function; for example for sig this happens when $|x| > 5$ because the gradient σ' is very small (see figure **??**). For multi-layer ANNs, such as DNNs, even if no single neuron saturates the activation function, due to the chain rule, weight updates for layers near the input potentially have products of very many factors that are less than one. Overfitting can be interpreted as memorization of the training samples; since the optimization (eqn. (**??**)) that leads to the Delta rule only measures $|t_k - y(\mathbf{x}_k)|$ DNNs can simply learn weights that encode responses to many (or most) of the training samples (since DNNs have so many weight parameters[6]).

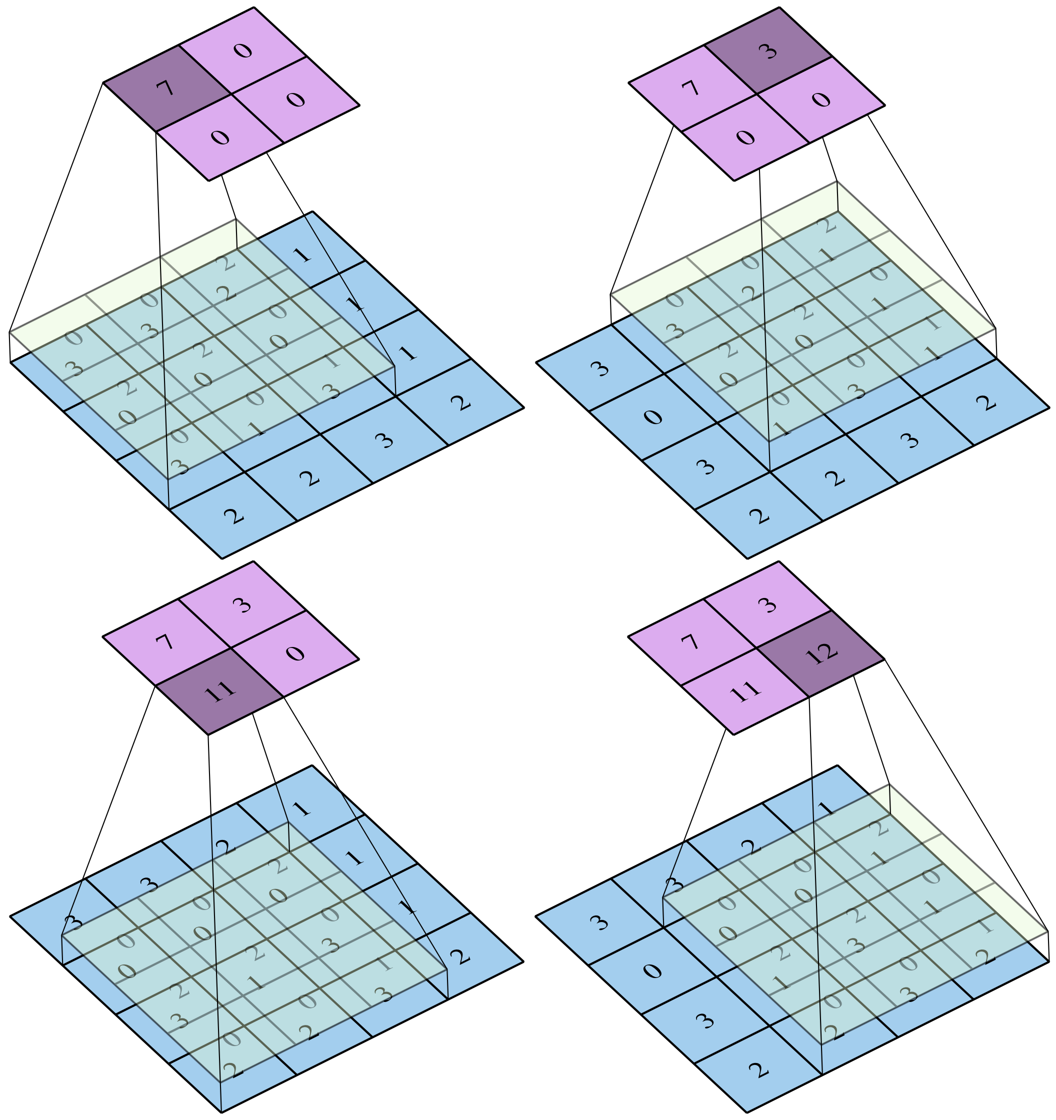
Deep Learning is a collection of techniques that make training Deep Neural Networks (DNNs) tractable. In

[5]To learn, in this context, means to effectively approximate a function that effectively performs a task. For example in the case of an object recognition task, learning entails approximating a function that outputs high values when an object is recognized and low values otherwise.

[6]OpenAI's Generative Pre-trained Transformer**radford2018improving** natural language processing (NLP) network has approximately 150 million parameters (and takes 8 GPU-months of non-stop training).

$$f = \begin{array}{|c|c|c|c|}\hline 3 & 3 & 2 & 1 \\ \hline 0 & 0 & 0 & 1 \\ \hline 3 & 1 & 3 & 1 \\ \hline 2 & 2 & 3 & 2 \\ \hline \end{array} \quad g = \begin{array}{|c|c|c|}\hline 0 & 0 & 2 \\ \hline 2 & 2 & 0 \\ \hline 0 & 0 & 1 \\ \hline \end{array} \quad f * g = \begin{array}{|c|c|}\hline 7 & 3 \\ \hline 11 & 12 \\ \hline \end{array}$$

(a) Input f, 2-D 3×3 convolution kernel g, and output $f * g$.



(b) Evaluation of convolution.

Fig. 4: 2-D convolution.

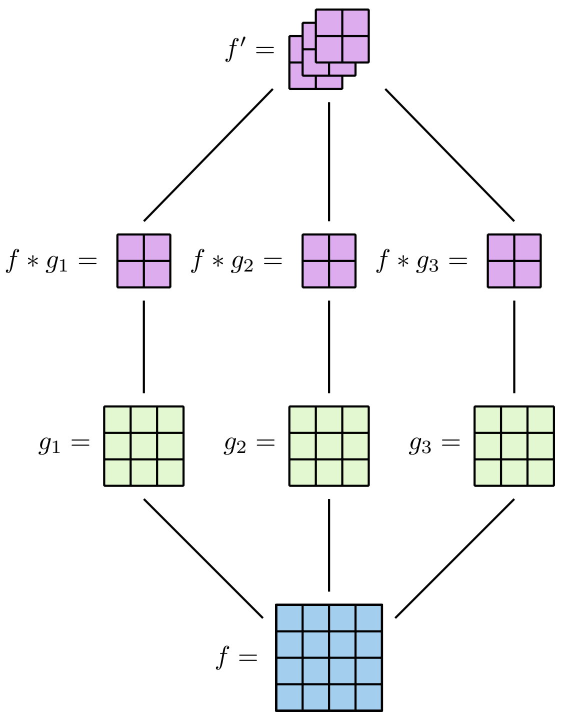


Fig. 5: A convolution layer consisting of three distinct 3×3 filters.

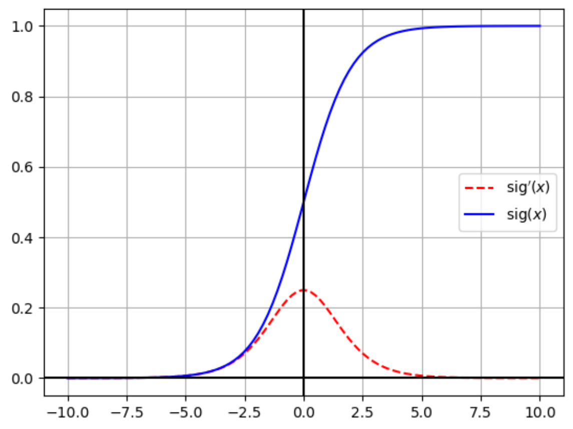
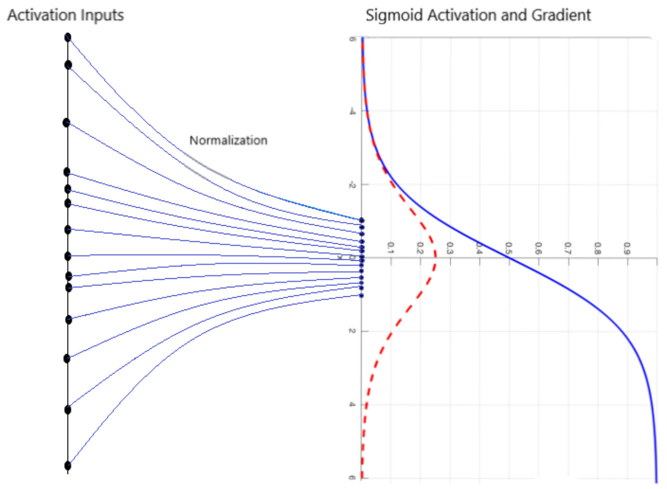


Fig. 6: Sigmoid activation gradients. Note that for $|x| > 5$ the gradient is almost 0.

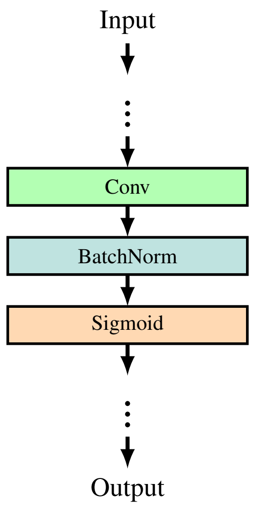
order to combat vanishing gradient *Batch Normalization***ioffe2015batch** is used to normalize outputs from linear layers prior to activation. On a batch by batch basis inputs to each layer's activation function (i.e., the activation functions of all of the neurons in the layer) are $(0,1)-$Normal normalized:

$$\boldsymbol{\mu}_B := \frac{1}{m}\sum_{j=1}^{m}\boldsymbol{x}_j \tag{16}$$

$$\boldsymbol{\sigma}_B^2 := \frac{1}{m}\sum_{j=1}^{m}(\boldsymbol{x}_j - \boldsymbol{\mu}_B)^2 \tag{17}$$



(a) Batch normalization's effect, for a single neuron, on inputs to a Sigmoid activation function.



(b) Batch normalization as a layer in a DNN.

Fig. 7: Batch normalization.

$$\hat{\boldsymbol{x}} := \frac{\boldsymbol{x} - \boldsymbol{\mu}_B}{\sqrt{\boldsymbol{\sigma}_B^2}} \tag{18}$$

where m is the batch size and the operations in eqn. (**??**) are understood to be broadcast (i.e., $\sqrt{\boldsymbol{\sigma}_B^2}$ is an element-wise root and elements of $\sqrt{\boldsymbol{\sigma}_B^2}$ divide corresponding elements of $\boldsymbol{x}-\boldsymbol{\mu}_B$). Overfitting is mititaged by using regularization; methods such as L_2 regularization and *dropout* have been shown to be effective against overfitting**bengio2013**. L_2 regularization introduces a squared weight term $\frac{\lambda}{2}\sum_{i=1}^{n} w_i^2$ to eqn. (**??**) which translates into a *weight decay* term λw_i^t in the Delta rule (eqn. (**??**))

$$\Delta w_i^t = \alpha \cdot \sum_j (t_j - y(\mathbf{x}_j)) \cdot \sigma' \cdot x_{ij} + \lambda w_i^t \tag{19}$$

Dropout effects regularization by selectively disabling neurons in the network (see figure **??**); on each forward pass through the network a neuron is either passed input or not according to some probability p (usually $p = 0.5$). The intuition being that not every neuron will have an opportunity to learn from every

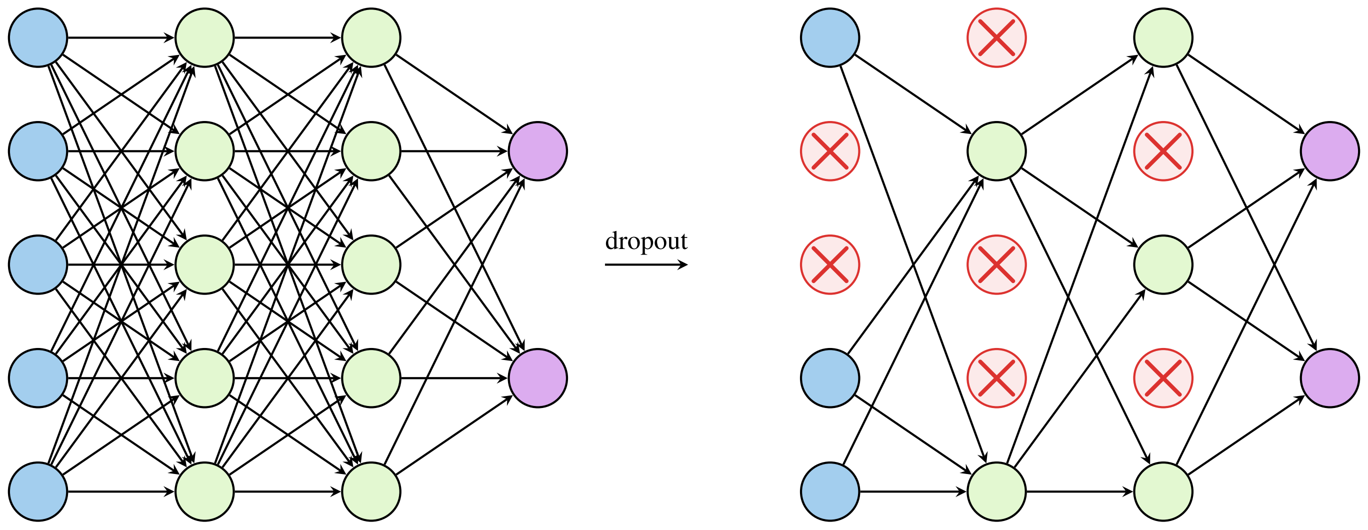
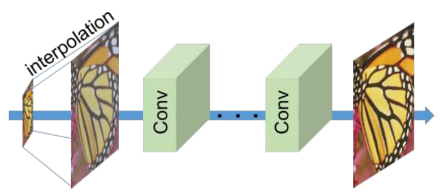
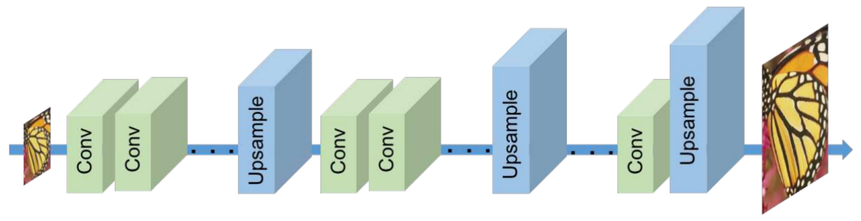


Fig. 8: Dropout regularization technique. Note that approximately only 50% of the nodes are active.

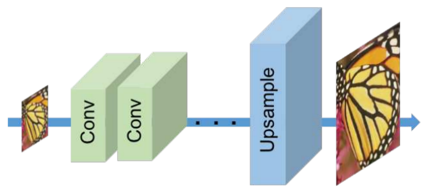
sample thereby limiting its, and the network's as a whole, ability to memorize patterns unique to the training samples (i.e., overfit).



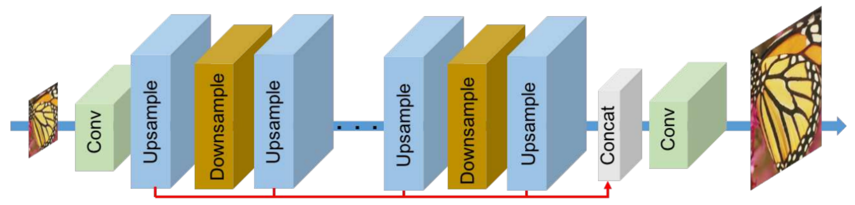
(a) Predefined up-sampling.



(b) Progressive up-sampling.



(c) Single up-sampling.



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

Fig. 9: Comparison of Deep SR network architectures**haris2018deep**