EE604 Class Note: Denoising Autoencoder

Tanaya Guha

So far in class, we have been talking about various denoising techniques (e.g. bilateral filtering, non-local means) that use hand-designed filters or kernels for image correction. Denoising autoencoder (Vincent et al., ICML 2008) however follows a completely different paradigm. It follows a completly data-driven approach where instead of designing a kernel we let the system *learn* them from data. A denoising autoencoder is a simple extension of the classical autoencoder, and was proposed as a fundamental building block for deep neural networks. An autoencoder is a neural network which learns to reconstruct its own input. A neural network, in turn, is an interconnected group of artificial neurons (computational units), whose development is inspired by the vast network of interconnected neurons in human brain.

1 A single neuron

The building block of a neural network is an artificial neuron (simply, a neuron). A neuron is a computational unit that performs a two-step computation: preactivation and activation. Let us consider a neuron which takes n inputs: $\mathbf{x} = [\mathrm{x}_1, \mathrm{x}_2, ..., \mathrm{x}_\mathrm{n}]$ (see Fig. 1). You may imagine these input values to be the pixel intensities of an image. Let there also be n weights $\mathrm{w}_1, \mathrm{w}_2, \mathrm{w}_3, ..., \mathrm{w}_\mathrm{n}$, where w_i is associated with the input x_i. The preactivation of a neuron is thus a weighted sum of its inputs: $\sum_{i=1}^{n} \mathrm{w}_\mathrm{i}\mathrm{x}_\mathrm{i}$. Usually, there is a *bias* term b which also gets added, yielding a preactivation term as $a(\mathbf{x}) = \mathrm{b} + \sum_{\mathrm{i}=1}^{\mathrm{n}} \mathrm{w}_\mathrm{i}\mathrm{x}_\mathrm{i}$. You may think of the bias term as a weight of an additional input which has a value of 1. The activation of a neuron is computed after passing $\mathrm{a}(\mathbf{x})$ through a known function g(.), called the *activation function*. Typically, g(.) is a non-linear function. Hence the final output of a neuron is given by

$$\mathrm{h}(\mathbf{x}) = \mathrm{g}(\mathrm{a}(\mathbf{x}))) = \mathrm{g}(\mathrm{b} + \sum_{\mathrm{i}=1}^{\mathrm{n}} \mathrm{w}_\mathrm{i}\mathrm{x}_\mathrm{i})$$

Some common choices for g(.) are as follows:

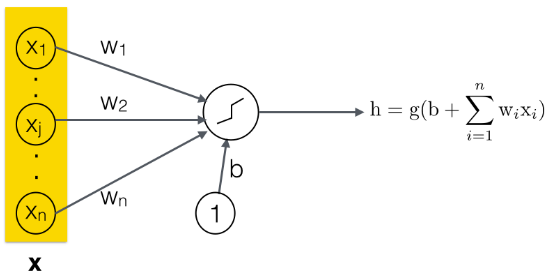


Figure 1: A single neuron

- **Linear**: $\mathrm{g}_{\mathrm{lin}}(\mathrm{a}) = \mathrm{a}$
  This is an identity mapping; although the simplest, often not very interesting.
- **Logistic/Sigmoid (logistic)**: $\mathrm{g}_{\mathrm{sig}}(\mathrm{a}) = \frac{1}{1+\mathrm{e}^{-\mathrm{a}}}$
  This function squashes the input between 0 and 1. Also note that the function is strictly increasing and bounded in both directions.
- **Hyperbolic tangent (TanH)**: $\mathrm{g}_{\mathrm{tanh}}(\mathrm{a}) = \frac{\mathrm{e}^{\mathrm{a}}-\mathrm{e}^{-\mathrm{a}}}{\mathrm{e}^{\mathrm{a}}+\mathrm{e}^{-\mathrm{a}}}$
  This function squashes the input between -1 and 1. Also, a bounded and strictly increasing function.

- **Rectified linear unit (ReLU)**: $g_{rlu}(a) = \max(0, a)$.
  The output of this function is non-negative, but is not upper-bounded. One advantage of using this activation is to achieve ‘sparse’ activation, i.e. when we want to activate only some neurons, and set others to zero.

The primary reason behind choosing certain activation functions lies in the convenience of computing their gradients. As we will see in the following sections, training a neural network involves computing gradients of these functions. See this link for other activation functions and more information.

`Exercise: Show that (i)` $g'_{sig}(a) = g_{sig}(a)(1 - g_{sig}(a))$, `and (ii)` $g'_{tanh}(a) = 1 - g^2_{tanh}(a)$

2 A simple neural network

Now that we have understood the operation of a single neuron, let’s see how we can organize them in layers to create a simple neural network. In its most basic form, a neural network consists of at least 3 layers: the input layer, at least one hidden layer, and an output layer (see Fig. 2). Apart from the input layer, all other layers contain neurons (note the symbol) with non-linear activation. Such networks are also called multi-layer perceptron (MLP).

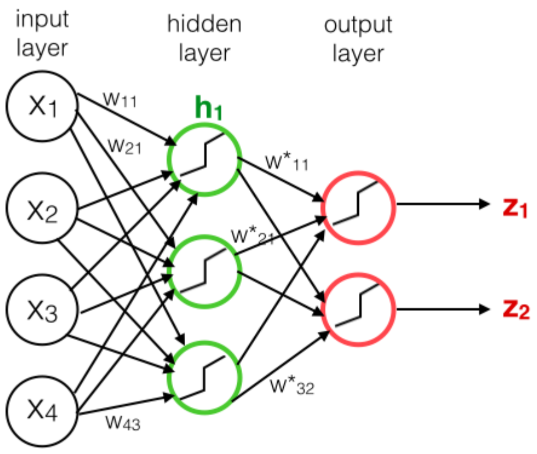


Figure 2: Multilayer perceptron

Let us consider the simple network in Fig. 2. It has an input layer, one hidden layer, and one output layer. Note that this is a fully connected network, which means that between consecutive layers, every node is connected to every other node. The input layer contains 4 nodes, where each node takes a single input value x_i, where $i = 1, 2, 3, 4$. The input nodes are *not* neurons; they simply copy the input values. The hidden layer has 3 nodes (neurons): h_1, h_2, h_3. We also have the weights associated with each input - w_{ji} is the weight corresponding to the edge connecting the hidden node h_j with the input node x_i. There are also the bias terms b_j, $j = 1, 2, 3$, one for each hidden node. For simplicity, the bias terms are omitted from Fig. 2.

Consider the node h_1 in the hidden layer. How many inputs is it connected to? Yes, all the 4 inputs x_1, x_2, x_3, x_4 weighted by $w_{11}, w_{12}, w_{13}, w_{14}$, respectively. The output of h_1 is thus given by $h_1(x) = g(b_1 + \sum_{i=1}^{4} w_{1i}x_i)$. Writing the output of the entire hidden layer in matrix-vector form we get the following equation.

$$\mathbf{h} = g_1(\mathbf{b} + \mathbf{W}\mathbf{x}) \tag{1}$$

Here, $\mathbf{h} = [h_1(x)\ \ h_2(x)\ \ h_3(x)]^T$ and $\mathbf{b} = [b_1\ \ b_2\ \ b_3]^T$ are vectors of dimension 3, $\mathbf{W} = \{w_{ji}\}$ ($j = 1, 2, 3$ and $i = 1, 2, 3, 4$) is the weight matrix of dimension 3×4, and $\mathbf{x}$ is the 4-dimensional input vector. Now $\mathbf{h}$ acts as the input to the output layer, which has 2 nodes in this case. Adopting matrix-vector notations, we can write the output equation as follows:

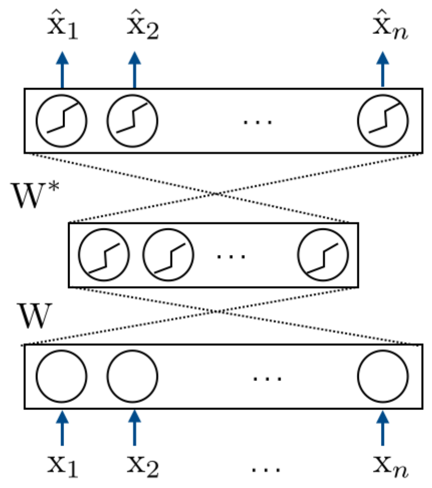
$$\mathbf{z} = g_2(\mathbf{b}^* + \mathbf{W}^*\mathbf{h}) \tag{2}$$

where $\mathbf{z} = [\mathrm{z}_1(\mathrm{x})\ \ \mathrm{z}_2(\mathrm{x})]^T$ and $\mathbf{b}^* = [\mathrm{b}_1^*\ \ \mathrm{b}_2^*]^T$ are vectors of dimension 2, $\mathbf{W}^* = \{\mathrm{w}_{ji}^*\}$ ($j = 1, 2, 3$ and $i = 1, 2, 3, 4$) contain the weights. Now $\mathbf{h}$ acts as the input to the output layer, which has 2 nodes in our case. Note that g_1 and g_2 need not be the same non-linearity function. Of course, there can be any number of nodes in a given layer depending on design and requirement. The *feedforward* equations (1) and (2) will hold. These equations are called feedforward because they are used to compute the output of a network given an input and known values of the parameters $\mathbf{W}, \mathbf{W}*, \mathbf{b}$ and $\mathbf{b}*$. These parameters are usually unknown. We first *train* a given network network using some known examples (i.e. known pairs of ($\mathbf{x}$ and$\mathbf{z}$ given to us, where $\mathbf{x}$ is called the training samples and $\mathbf{z}$ are corresponding labels). Training a neural network essentially means learning the network parameters so as to minimize a relevant loss function (will be described in the following sections). Once the training is complete, and we obtain the parameter values, the network can be used to predict the output labels ($\mathbf{z}$ values) for any new input.

Here we discussed only a simple neural network. A deep neural network will have many such hidden layers (and some other interesting features). Nevertheless, the basic mechanism remains the same.

3 Autoencoder

An autoencoder is a neural network which learns a mapping with itself, i.e. the input layer and the output layer are supposed to be the same. In its simplest form, an autoencoder has 3 layers: an input layer, a hidden layer and an output layer. Let the input be $\mathbf{x} = [\mathrm{x}_1, \mathrm{x}_2, ..., \mathrm{x}_\mathrm{n}]$, where the input to the i^{th} node is x_i. Let's say the hidden layer has m ($m \leq n$) nodes (neurons), and its output is given by $\mathbf{h} = \mathrm{g}_1(\mathbf{b} + \mathbf{W}\mathbf{x})$. This is often called the encoding step. The output layer of an autoencoder should have the same dimension as the input. The output of an autoencoder is given by $\hat{\mathbf{x}} = \mathrm{g}_2(\mathbf{b}^* + \mathbf{W}^*\mathbf{h})$. This is called the decoding step.



Training an autoencoder essentially means learning all the parameters of the network i.e. the weights and the biases. So during the training phase, our task is to learn $\mathbf{W}, \mathbf{W}*, \mathbf{b}$ and $\mathbf{b}*$, from a set of examples. A common practice is to set $W^* = W^T$ i.e. a case when the weights are tied. This way, the number of parameters to be learned are greatly reduced.

Let us now take a concrete example to see how the whole thing works. Let us consider a training set of $K = 100$ images $\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_K$, each of size 10×10. This will require us to build an autoencoder with an input and output layer of dimensionality 100. If the hidden layer has 50 nodes (say), then we will be learning 50×100 weights (with tied weights) and the biases. We start by randomly initializing the weights , and then computing the output values $\hat{\mathbf{x_1}}, \hat{\mathbf{x_2}}, ..., \hat{\mathbf{x_K}}$ by a *forward pass.* We define a simple squared error-based cost function as follows:

$$\mathcal{E} = \frac{1}{K}\sum_{i=1}^{K} \|\mathbf{x}_i - \hat{\mathbf{x}}_i\|_2^2 \tag{3}$$

A cross-entropy loss is also used in cases where the input is seen vectors of probabilities. This loss is defines as follows:

$$\mathcal{E}_H = -\sum_{i=1}^{n}[\mathbf{x}_i \log\hat{\mathbf{x}}_i + (1-\mathbf{x}_i)\log(1-\hat{\mathbf{x}}_i)] \tag{4}$$

Let's stick to the squared error as our loss function for now. Our objective is to minimize E over the weights $\mathbf{W}$ (and $\mathbf{b}$). This is usually done by some form of a *gradient descent* algorithm, e.g. stochastic gradient descent. Such algorithm gives us simple update rules which involves computing the gradient of E with respect to the weight. In particular, the update rule for a parameter $\mathrm{w} \in \mathbf{W}$ is given by

$$\mathrm{w} = \mathrm{w} - \alpha\frac{\partial\mathcal{E}}{\partial\mathrm{w}} \tag{5}$$

where α (a scalar) is called the step size (or learning rate in machine learning), which controls the amount of change allowed in every iteration. As you can easily see, the main task here is to compute the gradient. The error gradient (w.r.t. any weight) is computed by taking advantage of the chain rule. To understand this clearly, let us consider the toy network below. If we want

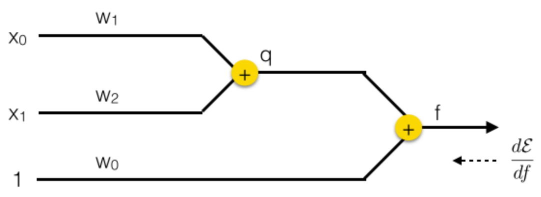


Figure 3: Toy backpropagation example

to update the weight w_2, we would like to compute $\frac{\partial\mathcal{E}}{\partial\mathrm{w}_2}$. Using chain rule, we can write $\frac{\partial\mathcal{E}}{\partial\mathrm{w}_2}$ as follows:

$$\frac{\partial\mathcal{E}}{\partial\mathrm{w}_2} = \frac{\partial\mathrm{q}}{\partial\mathrm{w}_2} \times \frac{\partial\mathrm{f}}{\partial\mathrm{q}} \times \frac{\partial\mathcal{E}}{\partial\mathrm{f}} \tag{6}$$

Note that the first two gradients in eq. (6) are the local gradients as they do not depend on the values in other parts of the network. The last derivative however can be computed only when the entire forward pass is complete. This gradient ($\frac{\partial\mathcal{E}}{\partial\mathrm{f}}$) is computed at the end, and then propagated all the way back to the branch whose weight we are going to update. This method of propagating the error gradient is called *backpropagation*. For step by step backpropagation examples, refer to your class note and/or visit this link. The algorithms used for training an autoencode (or any neural net for that matter) may vary depending on the number of inputs being processed or which particular update rule is used. Nevertheless, in all cases, the training stage is an iterative learning process where feedforward and backprop are performed back and forth to minimize the loss function. When the training is complete, the network can produce an output $\hat{\mathbf{x}}_{new}$ for a test input $\mathbf{x}_{new}$ by a forward pass.

4 Denoising autoencoder

The denoising autoencoder is an extension of the classical autoencoder. The difference here is that the input to the autoencoder is not the clean images, but a corrupted version of them. Let us consider our original input images $\mathbf{x}_i$, where $i = 1, 2, .., K$. Following the original work (Vincent et al., ICML 2008), a corrupted version of these images $\mathbf{x}_{corr_i}$ can be obtained by randomly setting a number of pixel values to zero. This is impulse noise if you remember (the method has been found to work well for Gaussian noise too). A denoising autoencoder takes these $\mathbf{x}_{corr_i}$ as inputs and minimizes a loss function of the following form:

$$\mathcal{E} = \frac{1}{K}\sum_{i=1}^{K}\|\mathbf{x} - \hat{\mathbf{x}}_i\|_2^2 \tag{7}$$

Note that the loss is still computed w.r.t. the clean images. The output $\mathbf{x}$ is expected to be clean. Thus the denoising autoencoder learns a hidden representation which is robust to distortions, and hence can produce a cleaner image. When the weight matrix $\mathbf{W}$ is visualized as small images (weights corresponding to each neuron can be visualized as a small image), edge-like structures appear (see (Vincent et al., ICML 2008) for a visualization).

Further reading

- http://deeplearning.net/tutorial/dA.html
- Bengio, Learning deep architectures for AI, *Foundations and Trends in Machine Learning 1(2)*, pages 1-127
- Vincent et al., "Extracting and composing robust features with denoising autoencoders", *Conference on Machine Learning (ICML'08)*, pages 1096 - 1103, 2008.
- https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/