CS294A Lecture notes

Andrew Ng

Sparse autoencoder

1 Introduction

Supervised learning is one of the most powerful tools of AI, and has led to automatic zip code recognition, speech recognition, self-driving cars, and a continually improving understanding of the human genome. Despite its significant successes, supervised learning today is still severely limited. Specifically, most applications of it still require that we manually specify the input features x given to the algorithm. Once a good feature representation is given, a supervised learning algorithm can do well. But in such domains as computer vision, audio processing, and natural language processing, there're now hundreds or perhaps thousands of researchers who've spent years of their lives slowly and laboriously hand-engineering vision, audio or text features. While much of this feature-engineering work is extremely clever, one has to wonder if we can do better. Certainly this labor-intensive hand-engineering approach does not scale well to new problems; further, ideally we'd like to have algorithms that can automatically learn even better feature representations than the hand-engineered ones.

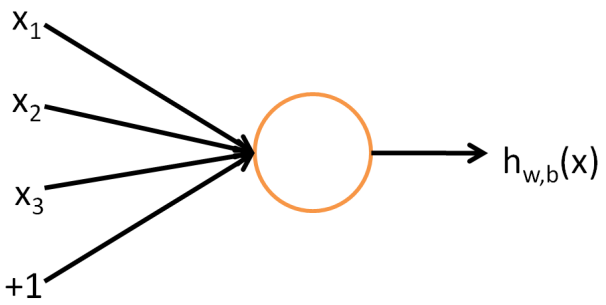
These notes describe the **sparse autoencoder** learning algorithm, which is one approach to automatically learn features from unlabeled data. In some domains, such as computer vision, this approach is not by itself competitive with the best hand-engineered features, but the features it can learn do turn out to be useful for a range of problems (including ones in audio, text, etc). Further, there're more sophisticated versions of the sparse autoencoder (not described in these notes, but that you'll hear more about later in the class) that do surprisingly well, and in some cases are competitive with or sometimes even better than some of the hand-engineered representations.

This set of notes is organized as follows. We will first describe feedforward neural networks and the backpropagation algorithm for supervised learning. Then, we show how this is used to construct an autoencoder, which is an unsupervised learning algorithm, and finally how we can build on this to derive a sparse autoencoder. Because these notes are fairly notation-heavy, the last page also contains a summary of the symbols used.

2 Neural networks

Consider a supervised learning problem where we have access to labeled training examples $(x^{(i)}, y^{(i)})$. Neural networks give a way of defining a complex, non-linear form of hypotheses $h_{W,b}(x)$, with parameters W, b that we can fit to our data.

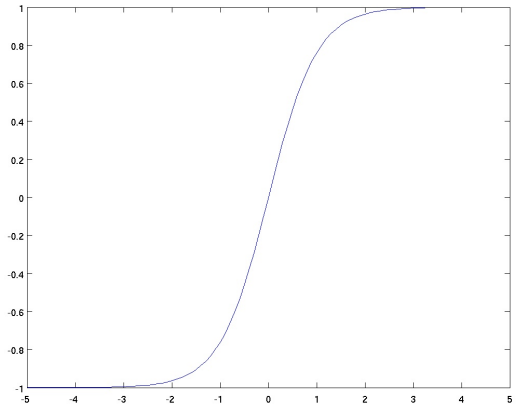
To describe neural networks, we use the following diagram to denote a single "neuron":



This "neuron" is a computational unit that takes as input x_1, x_2, x_3 (and a $+1$ intercept term), and outputs $h_{w,b}(x) = f(w^T x) = f(\sum_{i=1}^{3} w_i x_i + b)$, where $f : \mathbb{R} \mapsto \mathbb{R}$ is called the **activation function**. One possible choice for $f(\cdot)$ is the sigmoid function $f(z) = 1/(1 + \exp(-z))$; in that case, our single neuron corresponds exactly to the input-output mapping defined by logistic regression. In these notes, however, we'll use a different activation function, the hyperbolic tangent, or tanh, function:

$$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}, \tag{1}$$

Here's a plot of the $\tanh(z)$ function:



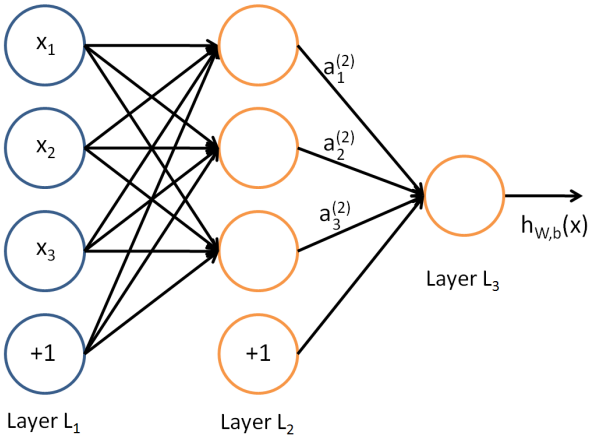
The $\tanh(z)$ function is a rescaled version of the sigmoid, and its output range is $[-1, 1]$ instead of $[0, 1]$. Our description of neural networks will use this activation function.

Note that unlike CS221 and (parts of) CS229, we are not using the convention here of $x_0 = 1$. Instead, the intercept term is handled separated by the parameter b.

Finally, one identity that'll be useful later: If $f(z) = \tanh(z)$, then its derivative is given by $f'(z) = 1 - (f(z))^2$. (Derive this yourself using the definition of $\tanh(z)$ given in Equation 1.)

2.1 Neural network formulation

A neural network is put together by hooking together many of our simple "neurons," so that the output of a neuron can be the input of another. For example, here is a small neural network:



In this figure, we have used circles to also denote the inputs to the network. The circles labeled "+1" are called **bias units**, and correspond to the intercept term. The leftmost layer of the network is called the **input layer**, and the rightmost layer the **output layer** (which, in this example, has only one node). The middle layer of nodes is called the **hidden layer**, because its values are not observed in the training set. We also say that our example neural network has 3 **input units** (not counting the bias unit), 3 **hidden units**, and 1 **output unit**.

We will let n_l denote the number of layers in our network; thus $n_l = 3$ in our example. We label layer l as L_l, so layer L_1 is the input layer, and layer L_{n_l} the output layer. Our neural network has parameters $(W, b) = (W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)})$, where we write $W_{ij}^{(l)}$ to denote the parameter (or weight) associated with the connection between unit j in layer l, and unit i in layer $l+1$. (Note the order of the indices.) Also, $b_i^{(l)}$ is the bias associated with unit i in layer $l+1$. Thus, in our example, we have $W^{(1)} \in \mathbb{R}^{3\times 3}$, and $W^{(2)} \in \mathbb{R}^{1\times 3}$. Note that bias units don't have inputs or connections going into them, since they always output the value +1. We also let s_l denote the number of nodes in layer l (not counting the bias unit).

We will write $a_i^{(l)}$ to denote the **activation** (meaning output value) of unit i in layer l. For $l = 1$, we also use $a_i^{(1)} = x_i$ to denote the i-th input. Given a fixed setting of the parameters W, b, our neural network defines a hypothesis $h_{W,b}(x)$ that outputs a real number. Specifically, the computation that this neural network represents is given by:

$$\begin{aligned} a_1^{(2)} &= f(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + b_1^{(1)}) && (2)\\ a_2^{(2)} &= f(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + b_2^{(1)}) && (3)\\ a_3^{(2)} &= f(W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3 + b_3^{(1)}) && (4)\\ h_{W,b}(x) &= a_1^{(3)} = f(W_{11}^{(2)}a_1 + W_{12}^{(2)}a_2 + W_{13}^{(2)}a_3 + b_1^{(2)}) && (5) \end{aligned}$$

In the sequel, we also let $z_i^{(l)}$ denote the total weighted sum of inputs to unit i in layer l, including the bias term (e.g., $z_i^{(2)} = \sum_{j=1}^{n} W_{ij}^{(1)}x_j + b_i^{(1)}$), so that $a_i^{(l)} = f(z_i^{(l)})$.

Note that this easily lends itself to a more compact notation. Specifically, if we extend the activation function $f(\cdot)$ to apply to vectors in an element-wise fashion (i.e., $f([z_1, z_2, z_3]) = [\tanh(z_1), \tanh(z_2), \tanh(z_3)]$), then we can

write Equations (2-5) more compactly as:

$$\begin{aligned} z^{(2)} &= W^{(1)}x + b^{(1)} \\ a^{(2)} &= f(z^{(2)}) \\ z^{(3)} &= W^{(2)}a^{(2)} + b^{(2)} \\ h_{W,b}(x) &= a^{(3)} = f(z^{(3)}) \end{aligned}$$

More generally, recalling that we also use $a^{(1)} = x$ to also denote the values from the input layer, then given layer l's activations $a^{(l)}$, we can compute layer $l+1$'s activations $a^{(l+1)}$ as:

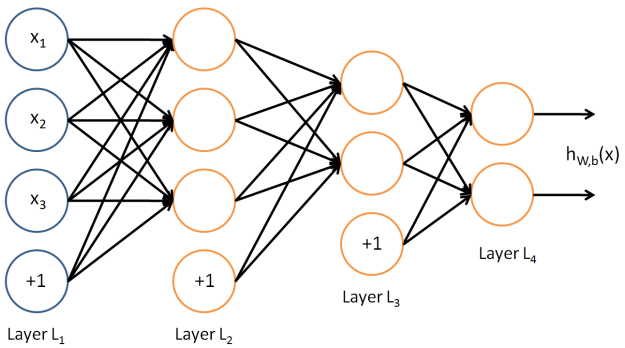
$$z^{(l+1)} = W^{(l)}a^{(l)} + b^{(l)} \tag{6}$$

$$a^{(l+1)} = f(z^{(l+1)}) \tag{7}$$

By organizing our parameters in matrices and using matrix-vector operations, we can take advantage of fast linear algebra routines to quickly perform calculations in our network.

We have so far focused on one example neural network, but one can also build neural networks with other **architectures** (meaning patterns of connectivity between neurons), including ones with multiple hidden layers. The most common choice is a n_l-layered network where layer 1 is the input layer, layer n_l is the output layer, and each layer l is densely connected to layer $l+1$. In this setting, to compute the output of the network, we can successively compute all the activations in layer L_2, then layer L_3, and so on, up to layer L_{n_l}, using Equations (6-7). This is one example of a **feedforward** neural network, since the connectivity graph does not have any directed loops or cycles.

Neural networks can also have multiple output units. For example, here is a network with two hidden layers layers L_2 and L_3 and two output units in layer L_4:



To train this network, we would need training examples $(x^{(i)}, y^{(i)})$ where $y^{(i)} \in \mathbb{R}^2$. This sort of network is useful if there're multiple outputs that you're interested in predicting. (For example, in a medical diagnosis application, the vector x might give the input features of a patient, and the different outputs y_i's might indicate presence or absence of different diseases.)

2.2 Backpropagation algorithm

We will train our neural network using stochastic gradient descent. For much of CS221 and CS229, we considered a setting in which we have a fixed training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$, and we ran either batch or stochastic gradient descent on that fixed training set. In these notes, will take an **online learning** view, in which we imagine that our algorithm has access to an unending sequence of training examples $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), (x^{(3)}, y^{(3)}), \ldots\}$. In practice, if we have only a finite training set, then we can form such a sequence by repeatedly visiting our fixed training set, so that the examples in the sequence will repeat. But even in this case, the online learning view will make some of our algorithms easier to describe. In this setting, stochastic gradient descent will proceed as follows:

For $i = 1, 2, 3, \ldots$

Get next training example $(x^{(i)}, y^{(i)})$.

Update $W_{jk}^{(l)} := W_{jk}^{(l)} - \alpha \frac{\partial}{\partial W_{jk}^{(l)}} J(W, b; x^{(i)}, y^{(i)})$

$b_j^{(l)} := b_j^{(l)} - \alpha \frac{\partial}{\partial b_j^{(l)}} J(W, b; x^{(i)}, y^{(i)})$

Here, α is the learning rate parameter, and $J(W, b) = J(W, b; x, y)$ is a cost function defined with respect to a single training example. (When there is no risk of ambiguity, we drop the dependence of J on the training example x, y, and simply write $J(W, b)$). If the training examples are drawn IID from some training distribution $\mathcal{D}$, we can think of this algorithm as trying to minimize

$$\mathrm{E}_{(x,y)\sim\mathcal{D}} \left[J(W, b; x, y) \right].$$

Alternatively, if our sequence of examples is obtained by repeating some fixed, finite training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$, then this algorithm is standard stochastic gradient descent for minimizing

$$\frac{1}{m} \sum_{i=1}^{m} J(W, b; x, y).$$

To train our neural network, we will use the cost function:

$$J(W,b;x,y) = \frac{1}{2}\left(\|h_{W,b}(x) - y\|^2\right) - \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(W_{ji}^{(l)}\right)^2$$

The first term is a sum-of-squares error term; the second is a regularization term (also called a **weight decay** term) that tends to decrease the magnitude of the weights, and helps prevent overfitting.[1] The **weight decay parameter** λ controls the relative importance of the two terms.

This cost function above is often used both for classification and for regression problems. For classification, we let $y = +1$ or -1 represent the two class labels (recall that the $\tanh(z)$ activation function outputs values in $[-1, 1]$, so we use $+1$/-1 valued outputs instead of 0/1). For regression problems, we first scale our outputs to ensure that they lie in the $[-1, 1]$ range.

Our goal is to minimize $\mathrm{E}_{(x,y)}[J(W,b;x,y)]$ as a function of W and b. To train our neural network, we will initialize each parameter $W_{ij}^{(l)}$ and each $b_i^{(l)}$ to a small random value near zero (say according to a $\mathcal{N}(0,\epsilon^2)$ distribution for some small ϵ, say 0.01), and then apply stochastic gradient descent. Since $J(W,b;x,y)$ is a non-convex function, gradient descent is susceptible to local optima; however, in practice gradient descent usually works fairly well. Also, in neural network training, stochastic gradient descent is almost always used rather than batch gradient descent. Finally, note that it is important to initialize the parameters randomly, rather than to all 0's. If all the parameters start off at identical values, then all the hidden layer units will end up learning the same function of the input (more formally, $W_{ij}^{(1)}$ will be the same for all values of i, so that $a_1^{(2)} = a_2^{(2)} = \ldots$ for any input x). The random initialization serves the purpose of **symmetry breaking**.

We now describe the **backpropagation** algorithm, which gives an efficient way to compute the partial derivatives we need in order to perform stochastic gradient descent. The intuition behind the algorithm is as follows. Given a training example (x, y), we will first run a "forward pass" to compute all the activations throughout the network, including the output value of the hypothesis $h_{W,b}(x)$. Then, for each node i in layer l, we would like to compute

[1] Usually weight decay is not applied to the bias terms $b_i^{(l)}$, as reflected in our definition for $J(W,b;x,y)$. Applying weight decay to the bias units usually makes only a small different to the final network, however. If you took CS229, you may also recognize weight decay this as essentially a variant of the Bayesian regularization method you saw there, where we placed a Gaussian prior on the parameters and did MAP (instead of maximum likelihood) estimation.

an "error term" $\delta_i^{(l)}$ that measures how much that node was "responsible" for any errors in our output. For an output node, we can directly measure the difference between the network's activation and the true target value, and use that to define $\delta_i^{(n_l)}$ (where layer n_l is the output layer). How about hidden units? For those, we will compute $\delta_i^{(l)}$ based on a weighted average of the error terms of the nodes that uses $a_i^{(l)}$ as an input. In detail, here is the backpropagation algorithm:

1. Perform a feedforward pass, computing the activations for layers L_2, L_3, and so on up to the output layer L_{n_l}.

2. For each output unit i in layer n_l (the output layer), set

$$\delta_i^{(n_l)} = \frac{\partial}{\partial z_i^{(n_l)}} \ \frac{1}{2} \left\| y - h_{W,b}(x) \right\|^2 = -(y_i - a_i^{(n_l)}) \cdot f'(z_i^{(n_l)})$$

3. For $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$

   For each node i in layer l, set

$$\delta_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) f'(z_i^{(l)})$$

4. Update each weight $W_{ij}^{(l)}$ and $b_j^{(l)}$ according to:

$$\begin{aligned} W_{ij}^{(l)} &:= W_{ij}^{(l)} - \alpha \left(a_j^{(l)} \delta_i^{(l+1)} + \lambda W_{ij}^{(l)} \right) \\ b_i^{(l)} &:= b_i^{(l)} - \alpha \delta_i^{(l+1)}. \end{aligned}$$

Although we have not proved it here, it turns out that

$$\begin{aligned} \frac{\partial}{\partial W_{ij}^{(l)}} J(W, b; x, y) &= a_j^{(l)} \delta_i^{(l+1)} + \lambda W_{ij}^{(l)}, \\ \frac{\partial}{\partial b_i^{(l)}} J(W, b; x, y) &= \delta_i^{(l+1)}. \end{aligned}$$

Thus, this algorithm is exactly implementing stochastic gradient descent. Finally, we can also re-write the algorithm using matrix-vectorial notation. We will use "$\bullet$" to denote the element-wise product operator (denoted "`.*`" in Matlab or Octave, and also called the Hadamard product), so that if

$a = b \bullet c$, then $a_i = b_i c_i$. Similar to how we extended the definition of $f(\cdot)$ to apply element-wise to vectors, we also do the same for $f'(\cdot)$ (so that $f'([z_1, z_2, z_3]) = [\frac{\partial}{\partial z_1} \tanh(z_1), \frac{\partial}{\partial z_2} \tanh(z_2), \frac{\partial}{\partial z_3} \tanh(z_3)]$). The algorithm can then be written:

1. Perform a feedforward pass, computing the activations for layers L_2, L_3, up to the output layer L_{n_l}, using Equations (6-7).

2. For the output layer (layer n_l), set

$$\delta^{(n_l)} = -(y - a^{(n_l)}) \bullet f'(z^{(n)})$$

3. For $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$

   Set

$$\delta^{(l)} = \left((W^{(l)})^T \delta^{(l+1)}\right) \bullet f'(z^{(l)})$$

4. Update the parameters according to:

$$\begin{aligned} W^{(l)} &:= W^{(l)} - \alpha \left(\delta^{(l+1)} (a^{(l)})^T + \lambda W^{(l)}\right) \\ b^{(l)} &:= b^{(l)} - \alpha \delta^{(l+1)}. \end{aligned}$$

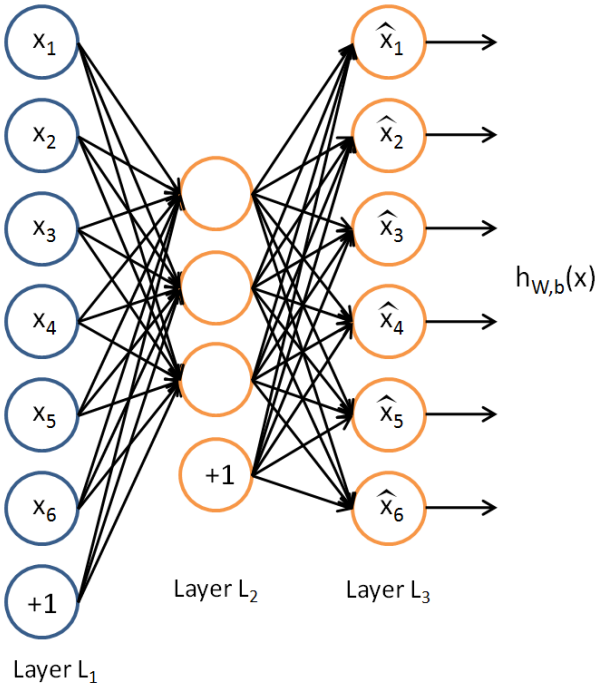
Implementation note 1: In steps 2 and 3 above, we need to compute $f'(z_i^{(l)})$ for each value of i. Assuming $f(z)$ is the tanh activation function, we would already have $a_i^{(l)}$ stored away from the forward pass through the network. Thus, using the expression that we worked out earlier for $f'(z)$, we can compute this as $f'(z_i^{(l)}) = 1 - (a_i^{(l)})^2$.

Implementation note 2: Backpropagation is a notoriously difficult algorithm to debug and get right, especially since many subtly buggy implementations of it—for example, one that has an off-by-one error in the indices and that thus only trains some of the layers of weights, or an implementation that omits the bias term, etc.—will manage to learn something that can look surprisingly reasonable (while performing less well than a correct implementation). Thus, even with a buggy implementation, it may not at all be apparent that anything is amiss. So, when implementing backpropagation, do read and re-read your code to check it carefully. Some people also numerically check their computation of the derivatives; if you know how to do this, it's worth considering too. (Feel free to ask us if you want to learn more about this.)

3 Autoencoders and sparsity

So far, we have described the application of neural networks to supervised learning, in which we are have labeled training examples. Now suppose we have only unlabeled training examples set $\{x^{(1)}, x^{(2)}, x^{(3)}, \ldots\}$, where $x^{(i)} \in \mathbb{R}^n$. An **autoencoder** neural network is an unsupervised learning algorithm that applies back propagation, setting the target values to be equal to the inputs. I.e., it uses $y^{(i)} = x^{(i)}$.

Here is an autoencoder:



The autoencoder tries to learn a function $h_{W,b}(x) \approx x$. In other words, it is trying to learn an approximation to the identity function, so as to output $\hat{x}$ that is similar to x. The identity function seems a particularly trivial function to be trying to learn; but by placing constraints on the network, such as by limiting the number of hidden units, we can discover interesting structure about the data. As a concrete example, suppose the inputs x are the pixel intensity values from a 10×10 image (100 pixels) so $n = 100$, and there are $s_2 = 50$ hidden units in layer L_2. Note that we also have $y \in \mathbb{R}^{100}$. Since there are only 50 hidden units, the network is forced to learn a *compressed* representation of the input. I.e., given only the vector of hidden unit activations $a^{(2)} \in \mathbb{R}^{50}$, it must try to **reconstruct** the 100-pixel

input x. If the input were completely random—say, each x_i comes from an IID Gaussian independent of the other features—then this compression task would be very difficult. But if there is structure in the data, for example, if some of the input features are correlated, then this algorithm will be able to discover some of those correlations.[2]

Our argument above relied on the number of hidden units s_2 being small. But even when the number of hidden units is large (perhaps even greater than the number of input pixels), we can still discover interesting structure, by imposing other constraints on the network. In particular, if we impose a **sparsity** constraint on the hidden units, then the autoencoder will still discover interesting structure in the data, even if the number of hidden units is large.

Informally, we will think of a neuron as being "active" (or as "firing") if its output value is close to 1, or as being "inactive" if its output value is close to -1. We would like to constrain the neurons to be inactive most of the time.[3]

We will do this in an online learning fashion. More formally, we again imagine that our algorithm has access to an unending sequence of training examples $\{x^{(1)}, x^{(2)}, x^{(3)}, \ldots\}$ drawn IID from some distribution $\mathcal{D}$. Also, let $a_i^{(2)}$ as usual denote the activation of hidden unit i in the autoencoder. We would like to (approximately) enforce the constraint that

$$\mathrm{E}_{x\sim\mathcal{D}}\left[a_i^{(2)}\right] = \rho,$$

where ρ is our **sparsity parameter**, typically a value slightly above -1.0 (say, $\rho \approx -0.9$). In other words, we would like the expected activation of each hidden neuron i to be close to -0.9 (say). To satisfy this expectation constraint, the hidden unit's activations must mostly be near -1.

Our algorithm for (approximately) enforcing the expectation constraint will have two major components: First, for each hidden unit i, we will keep a running estimate of $\mathrm{E}_{x\sim\mathcal{D}}\left[a_i^{(2)}\right]$. Second, after each iteration of stochastic gradient descent, we will slowly adjust that unit's parameters to make this expected value closer to ρ.

[2]In fact, this simple autoencoder often ends up learning a low-dimensional representation very similar to PCA's.

[3]The term "sparsity" comes from an alternative formulation of these ideas using networks with a sigmoid activation function f, so that the activations are between 0 or 1 (rather than -1 and 1). In this case, "sparsity" refers to most of the activations being near 0.

In each iteration of gradient descent, when we see each training input x we will compute the hidden units' activations $a_i^{(2)}$ for each i. We will keep a running estimate $\hat{\rho}_i$ of $\mathrm{E}_{x\sim\mathcal{D}}\left[a_i^{(2)}\right]$ by updating:

$$\hat{\rho}_i := 0.999\hat{\rho}_i + 0.001a_i^{(2)}.$$

(Or, in vector notation, $\hat{\rho} := 0.999\hat{\rho} + 0.001a^{(2)}$.) Here, the "0.999" (and "0.001") is a parameter of the algorithm, and there is a wide range of values will that will work fine. This particular choice causes $\hat{\rho}_i$ to be an exponentially-decayed weighted average of about the last 1000 observed values of $a_i^{(2)}$. Our running estimates $\hat{\rho}_i$'s can be initialized to 0 at the start of the algorithm.

The second part of the algorithm modifies the parameters so as to try to satisfy the expectation constraint. If $\hat{\rho}_i > \rho$, then we would like hidden unit i to become less active, or equivalently, for its activations to become closer to -1. Recall that unit i's activation is

$$a_i^{(2)} = f\left(\sum_{j=1}^{n} W_{ij}^{(1)} x_j + b_i^{(1)}\right), \tag{8}$$

where $b_i^{(1)}$ is the bias term. Thus, we can make unit i less active by decreasing $b_i^{(1)}$. Similarly, if $\hat{\rho}_i < \rho$, then we would like unit i's activations to become larger, which we can do by increasing $b_i^{(1)}$. Finally, the further ρ_i is from ρ, the more aggressively we might want to decrease or increase $b_i^{(1)}$ so as to drive the expectation towards ρ. Concretely, we can use the following learning rule:

$$b_i^{(1)} := b_i^{(1)} - \alpha\beta(\hat{\rho}_i - \rho) \tag{9}$$

where β is an additional learning rate parameter.

To summarize, in order to learn a sparse autoencoder using online learning, upon getting an example x, we will (i) Run a forward pass on our network on input x, to compute all units' activations; (ii) Perform one step of stochastic gradient descent using backpropagation; (iii) Perform the updates given in Equations (8-9).

4 Visualization

Having trained a (sparse) autoencoder, we would now like to visualize the function learned by the algorithm, to try to understand what it has learned.

Consider the case of training an autoencoder on 10×10 images, so that $n = 100$. Each hidden unit i computes a function of the input:

$$a_i^{(2)} = f\left(\sum_{j=1}^{100} W_{ij}^{(1)} x_j + b_i^{(1)}\right).$$

We will visualize the function computed by hidden unit i—which depends on the parameters $W_{ij}^{(1)}$ (ignoring the bias term for now) using a 2D image. In particular, we think of $a_i^{(1)}$ as some non-linear feature of the input x. We ask: What input image x would cause $a_i^{(1)}$ to be maximally activated? For this question to have a non-trivial answer, we must impose some constraints on x. If we suppose that the input is norm constrained by $||x||^2 = \sum_{i=1}^{100} x_i^2 \leq 1$, then one can show (try doing this yourself) that the input which maximally activates hidden unit i is given by setting pixel x_j (for all 100 pixels, $j = 1, \ldots, 100$) to

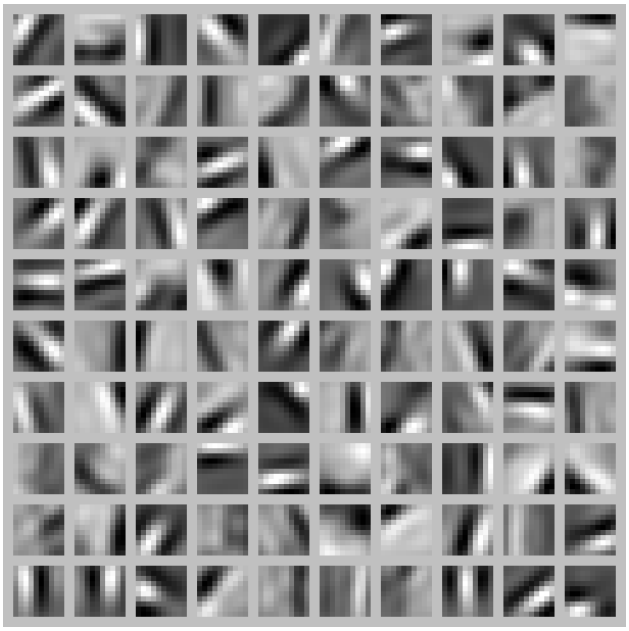
$$x_j = \frac{W_{ij}^{(1)}}{\sqrt{\sum_{j=1}^{100} (W_{ij}^{(1)})^2}}.$$

By displaying the image formed by these pixel intensity values, we can begin to understand what feature hidden unit i is looking for.

If we have an autoencoder with 100 hidden units (say), then we our visualization will have 100 such images—one per hidden unit. By examining these 100 images, we can try to understand what the ensemble of hidden units is learning.

When we do this for a sparse autoencoder (trained with 100 hidden units on 10x10 pixel inputs[4]) we get the following result:

[4]The results below were obtained by training on **whitened** natural images. Whitening is a preprocessing step which removes redundancy in the input, by causing adjacent pixels to become less correlated.



Each square in the figure above shows the (norm bounded) input image x that maximally actives one of 100 hidden units. We see that the different hidden units have learned to detect edges at different positions and orientations in the image.

These features are, not surprisingly, useful for such tasks as object recognition and other vision tasks. When applied to other input domains (such as audio), this algorithm also learns useful representations/features for those domains too.

5 Summary of notation

x	Input features for a training example, $x \in \mathbb{R}^n$.
y	Output/target values. Here, y can be vector valued. In the case of an autoencoder, $y = x$.
$(x^{(i)}, y^{(i)})$	The i-th training example
$h_{W,b}(x)$	Output of our hypothesis on input x, using parameters W, b. This should be a vector of the same dimension as the target value y.
$W_{ij}^{(l)}$	The parameter associated with the connection between unit j in layer l, and unit i in layer $l+1$.
$b_i^{(l)}$	The bias term associated with unit i in layer $l+1$. Can also be thought of as the parameter associated with the connection between the bias unit in layer l and unit i in layer $l+1$.
$a_i^{(l)}$	Activation (output) of unit i in layer l of the network. In addition, since layer L_1 is the input layer, we also have $a_i^{(1)} = x_i$.
$f(\cdot)$	The activation function. Throughout these notes, we used $f(z) = \tanh(z)$.
$z_i^{(l)}$	Total weighted sum of inputs to unit i in layer l. Thus, $a_i^{(l)} = f(z_i^{(l)})$.
α	Learning rate parameter
s_l	Number of units in layer l (not counting the bias unit).
n_l	Number layers in the network. Layer L_1 is usually the input layer, and layer L_{n_l} the output layer.
λ	Weight decay parameter
$\hat{x}$	For an autoencoder, its output; i.e., its reconstruction of the input x. Same meaning as $h_{W,b}(x)$.
ρ	Sparsity parameter, which specifies our desired level of sparsity
$\hat{\rho}_i$	Our running estimate of the expected activation of unit i (in the sparse autoencoder).
β	Learning rate parameter for algorithm trying to (approximately) satisfy the sparsity constraint.