Contents

UFLDL Tutorial

Description: This tutorial will teach you the main ideas of Unsupervised Feature Learning and Deep Learning. By working through it, you will also get to implement several feature learning/deep learning algorithms, get to see them work for yourself, and learn how to apply/adapt these ideas to new problems.

This tutorial assumes a basic knowledge of machine learning (specifically, familiarity with the ideas of supervised learning, logistic regression, gradient descent). If you are not familiar with these ideas, we suggest you go to this Machine Learning course and complete sections II, III, IV (up to Logistic Regression) first.

Supervised Learning and Optimization

Linear Regression

Problem Formulation

As a refresher, we will start by learning how to implement linear regression. The main idea is to get familiar with objective functions, computing their gradients and optimizing the objectives over a set of parameters. These basic tools will form the basis for more sophisticated algorithms later. Readers that want additional details may refer to the CS229 Lecture Notes on Supervised Learning for more.

Our goal in linear regression is to predict a target value y starting from a vector of input values $x \in \Re^n$. For example, we might want to make predictions about the price of a house so that y represents the price of the house in dollars and the elements x_j of x represent "features" that describe the house (such as its size and the number of bedrooms). Suppose that we are given many examples of houses where the features for the i'th house are denoted $x^{(i)}$ and the price is$y^{(i)}$. For short, we will denote the

Our goal is to find a function $y = h(x)$ so that we have $y^{(i)} \approx h(x^{(i)})$ for each training example. If we succeed in finding a function $h(x)$ like this, and we have seen enough examples of houses and their prices, we hope that the function $h(x)$ will also be a good predictor of the house price even when we are given the features for a new house where the price is not known.

To find a function $h(x)$ where $y^{(i)} \approx h(x^{(i)})$ we must first decide how to represent the function $h(x)$. To start out we will use linear functions:

$$h_\theta(x) = \sum_j \theta_j x_j = \theta^\top x \quad .$$

Here, $h_\theta(x)$ represents a large family of functions parametrized by the choice of θ. (We call this space of functions a "hypothesis class".) With this representation for h, our task is to find a choice of θ so that $h_\theta(x^{(i)})$ is as close as possible to $y^{(i)}$. In particular, we will search for a choice ofθ that minimizes:

$$J(\theta) = \frac{1}{2}\sum_i \left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 = \frac{1}{2}\sum_i \left(\theta^\top x^{(i)} - y^{(i)}\right)^2$$

This function is the "cost function" for our problem which measures how much error is incurred in predicting $y^{(i)}$ for a particular choice of θ. This may also be called a "loss", "penalty" or "objective" function.

Function Minimization

We now want to find the choice of θ that minimizes $J(\theta)$ as given above. There are many algorithms for minimizing functions like this one and we will describe some very effective ones that are easy to implement yourself in a later section (Gradient descent). For now, let's take for granted the fact that most commonly-used algorithms for function minimization require us to provide two pieces of information about $J(\theta)$: We will need to write code to compute $J(\theta)$and $\nabla_\theta J(\theta)$ on demand for any choice of θ. After that, the rest of the optimization procedure to find the best choice of / *theta* will be handled by the optimization algorithm. (Recall that the gradient $\nabla_\theta J(\theta)$ of a differentiable function J is a vector that points in the direction of steepest increase as a function of θ --- so it is easy to see how an optimization algorithm could use this to make a small change to θ that decreases (or increase) $J(\theta)$).

The above expression for $J(\theta)$ given a training set of $x^{(i)}$ and $y^{(i)}$ is easy to implement in MATLAB to compute $J(\theta)$ for any choice of θ. The remaining requirement is to compute the gradient:

$$\nabla_\theta J(\theta) = \begin{bmatrix} \frac{\partial J(\theta)}{\partial \theta_1} \\ \frac{\partial J(\theta)}{\partial \theta_2} \\ \vdots \\ \frac{\partial J(\theta)}{\partial \theta_n} \end{bmatrix}$$

Differentiating the cost function $J(\theta)$ as given above with respect to a particular parameter θ_j gives us:

$$\frac{\partial J(\theta)}{\partial \theta_j} = \sum_i x_j^{(i)} \left(h_\theta(x^{(i)}) - y^{(i)}\right).$$

Exercise 1A: Linear Regression

Starter code for this exercise is included in the [Starter code GitHub repo] in the ex1/ directory.

In this exercise you will implement the objective function and gradient calculations for linear regression in MATLAB.

In the ex1 directory of the starter code package you will find the file "ex1_linreg.m" which contains the makings of a simple linear regression experiment. This file performs most of the boiler-plate steps for you:

1. The data is loaded from "housing.data". An extra '1' feature is added to the dataset so that θ_1 will act as an intercept term in the linear function.
2. The examples in the dataset are randomly shuffled and the data is then split into a training and testing set. The features that are used as input to the learning algorithm are stored in the variables "train.X" and "test.X". The target value to be predicted is the estimated house price for each example. The prices are stored in "train.y" and "test.y", respectively, for the training and testing examples. You will use the training set to find the best choice of θ for predicting the house prices and then check its performance on the testing set.
3. The code calls the minFunc optimization package. minFunc will attempt to find the best choice of θ by minimizing the objective function implemented in linear_regression.m. It will be your job to implement linear_regression.m to compute the objective function value and the gradient with respect to the parameters.
4. After minFunc completes (i.e., after training is finished), the training and testing error is printed out. Optionally, it will plot a quick visualization of the predicted and actual prices for the examples in the test set.

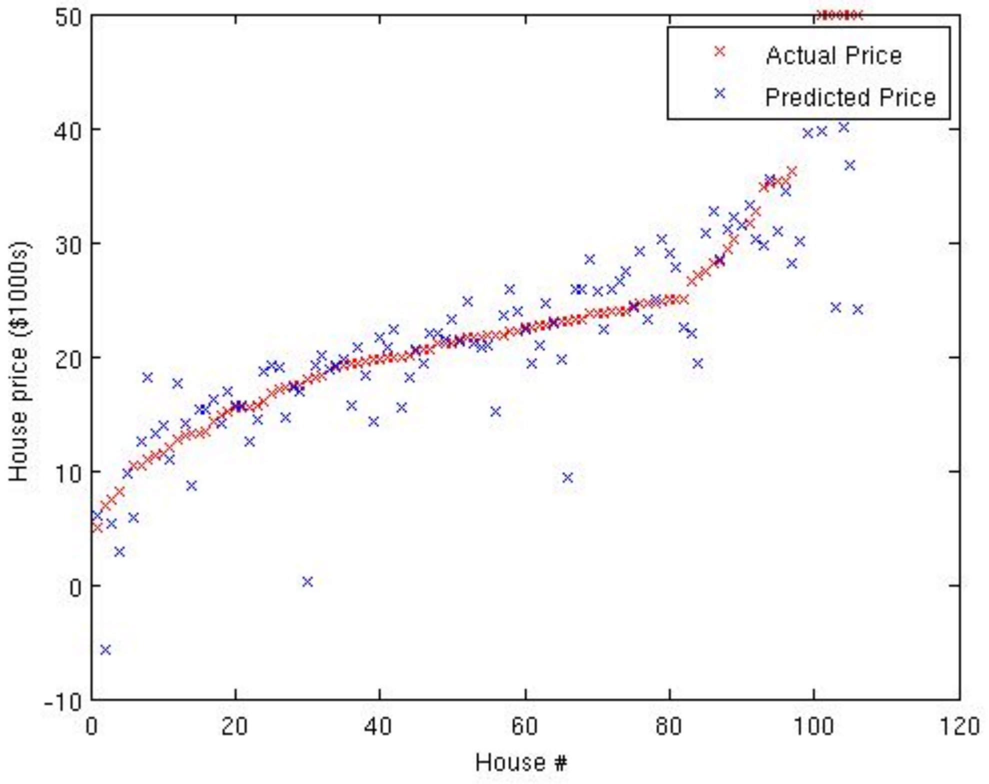
The ex1_linreg.m file calls the linear_regression.m file that must be filled in with your code. The linear_regression.m file receives the training data X, the training target values (house prices) y, and the current parameters θ.

Complete the following steps for this exercise:

1. Fill in the linear_regression.m file to compute $J(\theta)$ for the linear regression problem as defined earlier. Store the computed value in the variable 'f'.
2. Also in linear_regression.m, compute the gradient $\nabla_\theta J(\theta)$ and store it in the variable 'g'. That is, upon completion, the value of g(i) should contain $\frac{\partial J(\theta)}{\partial \theta_i}$.

You may complete both of these steps by looping over the examples in the training set (the columns of the data matrix X) and, for each one, adding its contribution to 'f' and 'g'. We will create a faster version in the next exercise.

Once you complete the exercise successfully, the resulting plot should look something like the one below:



(Yours may look slightly different depending on the random choice of training and testing sets.) Typical values for the RMS training and testing error are between 4.5 and 5.

Logistic Regression

Previously we learned how to predict continuous-valued quantities (e.g., housing prices) as a linear function of input values (e.g., the size of the house). Sometimes we will instead wish to predict a discrete variable such as predicting whether a grid of pixel intensities represents a "0" digit or a "1" digit. This is a classification problem. Logistic regression is a simple classification algorithm for learning to make such decisions.

In linear regression we tried to predict the value of $y^{(i)}$ for the i'th example $x^{(i)}$ using a linear function $y = h_\theta(x) = \theta^\top x.$. This is clearly not a great solution for predicting binary-valued labels ($y^{(i)} \in \{0, 1\}$). In logistic regression we use a different hypothesis class to try to predict the probability that a given example belongs to the "1" class versus the probability that it belongs to the "0" class. Specifically, we will try to learn a function of the form:

$$\begin{aligned}
P(y = 1|x) &= h_\theta(x) = \frac{1}{1 + \exp(-\theta^\top x)} \equiv \sigma(\theta^\top x), \\
P(y = 0|x) &= 1 - P(y = 1|x) = 1 - h_\theta(x).
\end{aligned}$$

The function

$$\sigma(z) \equiv \frac{1}{1 + \exp(-z)}$$

is often called the "sigmoid" or "logistic" function --- it is an S-shaped function that "squashes" the value of $\theta^\top x$ into the range [0,1] so that we may interpret $h_\theta(x)$ as a probability. Our goal is to search for a value of θ so that the probability $P(y = 1 \mid x) = h_\theta(x)$ is large when $x$belongs to the "1" class and small when x belongs to the "0" class (so that $P(y = 0 \mid x)$ is large). For a set of training examples with binary labels $\{(x^{(i)}, y^{(i)}) : i = 1, \ldots, m\}$ the following cost function measures how well a given h_θ does this:

$$J(\theta) = -\sum_i \left(y^{(i)} \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})) \right).$$

Note that only one of the two terms in the summation is non-zero for each training example (depending on whether the label $y^{(i)}$ is 0 or 1). When $y^{(i)} = 1$minimizing the cost function means we need to make $h_\theta(x^{(i)})$ large, and when $y^{(i)} = 0$ we want to make $1 - h_\theta$ large as explained above. For a full explanation of logistic regression and how this cost function is derived, see the CS229 notes on supervised learning.

We now have a cost function that measures how well a given hypothesis h_θ fits our training data. We can learn to classify our training data by minimizing $J(\theta)$to find the best choice of θ. Once we have done so, we can classify a new test point as "1" or "0" by checking which of these two class labels is most probable: if $P(y = 1 \mid x) > P(y = 0 \mid x)$ then we label the example as a "1", and "0" otherwise. This is the same as checking whether $h_\theta(x) > 0.5$.

To minimize $J(\theta)$ we can use the same tools as for linear regression. We need to provide a function that computes $J(\theta)$ and $\nabla_\theta J(\theta)$ for any requested choice of θ. The derivative of $J(\theta)$ as given above with respect to θ_j is:

$$\frac{\partial J(\theta)}{\partial \theta_j} = \sum_i x_j^{(i)} (h_\theta(x^{(i)}) - y^{(i)}).$$

Written in its vector form, the entire gradient can be expressed as:

$$\nabla_\theta J(\theta) = \sum_i x^{(i)} (h_\theta(x^{(i)}) - y^{(i)})$$

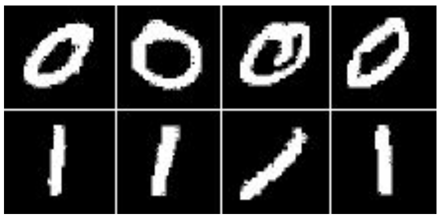
This is essentially the same as the gradient for linear regression except that now

$$h_\theta(x) = \sigma(\theta^\top x).$$

Exercise 1B

Starter code for this exercise is included in the [Starter code GitHub repo] in the ex1/ directory.

In this exercise you will implement the objective function and gradient computations for logistic regression and use your code to learn to classify images of digits from the MNIST dataset as either "0" or "1". Some examples of these digits are shown below:



Each of the digits is is represented by a 28x28 grid of pixel intensities, which we will reformat as a vector $x^{(i)}$ with 28*28 = 784 elements. The label is binary, so $y^{(i)} \in \{0,1\}$.

You will find starter code for this exercise in the ex1/ex1b_logreg.m file. The starter code file performs the following tasks for you:

1. Calls ex1_load_mnist.m to load the MNIST training and testing data. In addition to loading the pixel values into a matrix X (so that that j'th pixel of the i'th example is $X_{ji} = x_j^{(i)}$) and the labels into a row-vector y, it will also perform some simple normalizations of the pixel intensities so that they tend to have zero mean and unit variance. Even though the MNIST dataset contains 10 different digits (0-9), in this exercise we will only load the 0 and 1 digits --- the ex1_load_mnist function will do this for you.
2. The code will append a row of 1's so that θ_0 will act as an intercept term.
3. The code calls minFunc with the logistic_regression.m file as objective function. Your job will be to fill in logistic_regression.m to return the objective function value and its gradient.
4. After minFunc completes, the classification accuracy on the training set and test set will be printed out.

As for the linear regression exercise, you will need to implement logistic_regression.m to loop over all of the training examples $x^{(i)}$ and compute the objective$J(\theta;X,y)$. Store the resulting objective value into the variable f. You must also compute the gradient $\nabla_\theta J(\theta;X,y)$ and store it into the variable g. Once you have completed these tasks, you will be able to run the ex1b_logreg.m script to train the classifier and test it.

If your code is functioning correctly, you should find that your classifier is able to achieve 100% accuracy on both the training and testing sets! It turns out that this is a relatively easy classification problem because 0 and 1 digits tend to look very different. In future exercises it will be much more difficult to get perfect results like this.

Vectorization

For small jobs like the housing prices data we used for linear regression, your code does not need to be extremely fast. However, if your implementation for Exercise 1A or 1B used a for-loop as suggested, it is probably too slow to work well for large problems that are more interesting. This is because looping over the examples (or any other elements) sequentially in MATLAB is slow. To avoid for-loops, we want to rewrite our code to make use of optimized vector and matrix operations so that MATLAB will execute it quickly. (This is also useful for other languages, including Python and C/C++ --- we want to re-use optimized operations when possible.)

Following are some examples for how to vectorize various operations in MATLAB.

Example: Many matrix-vector products

Frequently we want to compute matrix-vector products for many vectors at once, such as when we compute $\theta^\top x^{(i)}$ for each example in a dataset (where θ may be a 2D matrix, or a vector itself). We can form a matrix X containing our entire dataset by concatenating the examples $x^{(i)}$ to form the columns of X:

$$X = \left[\begin{array}{cccc} | & | & | & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & | & | \end{array}\right]$$

With this notation, we can compute $y^{(i)} = Wx^{(i)}$ for all $x^{(i)}$ at once as:

$$\left[\begin{array}{cccc} | & | & | & | \\ y^{(1)} & y^{(2)} & \cdots & y^{(m)} \\ | & | & | & | \end{array}\right] = Y = WX$$

So, when performing linear regression, we can use $\theta^\top X$ to avoid looping over all of our examples to compute

$$y^{(i)} = \theta^\top X^{(i)}.$$

Example: normalizing many vectors

Suppose we have many vectors $x^{(i)}$ concatenated into a matrix X as above, and we want to compute $y^{(i)} = x^{(i)} / ||x^{(i)}||_2$ for all of the $x^{(i)}$. This may be done using several of MATLAB's array operations:

```
X_norm = sqrt( sum(X.^2,1) );
Y = bsxfun(@rdivide, X, X_norm);
```

This code squares all of the elements of X, then sums along the first dimension (the rows) of the result, and finally takes the square root of each element. This leaves us with a 1-by-m matrix containing $||x^{(i)}||_2$. The bsxfun routine can be thought of as expanding or cloning *X*norm so that it has the same dimension as X before applying an element-wise binary function. In the example above it divides every element $X_{ji} = x_j^{(i)}$ by the corresponding column in *X*norm, leaving us with

$$Y_{ji} = X_{ji}/X\mathrm{norm}_i = x_j^{(i)}/||x^{(i)}||_2$$

as desired. bsxfun can be used with almost any binary element-wise function (e.g., @plus, @ge, or @eq). See the bsxfun docs!

Example: matrix multiplication in gradient computations

In our linear regression gradient computation, we have a summation of the form:

$$\frac{\partial J(\theta; X, y)}{\partial \theta_j} = \sum_i x_j^{(i)}(\hat{y}^{(i)} - y^{(i)}).$$

Whenever we have a summation over a single index (in this case i) with several other fixed indices (in this case j) we can often rephrase the computation as a matrix multiply since

$$[AB]_{jk} = \sum_i A_{ji}B_{ik}$$

. If y and $\hat{y}$ are column vectors (so $y_i \equiv y^{(i)}$), then with this template we can rewrite the above summation as:

$$\frac{\partial J(\theta; X, y)}{\partial \theta_j} = \sum_i X_{ji}(\hat{y}_i - y_i) = [X(\hat{y} - y)]_j.$$

Thus, to perform the entire computation for every *j* we can just compute $X(\hat{y} - y)$.

In MATLAB:

```
% X(j,i) = j'th coordinate of i'th example.
% y(i) = i'th value to be predicted;    y is a column vector.
% theta = vector of parameters

y_hat = theta'*X; % so y_hat(i) = theta' * X(:,i).    Note that y_hat is a *row-vector*.
g = X*(y_hat' - y);
```

Exercise 1A and 1B Redux

Go back to your Exercise 1A and 1B code. In the ex1a_linreg.m file and ex1b_logreg.m file you will find commented-out code that calls minFunc using linear_regression_vec.m and logistic_regression_vec.m (respectively) instead of linear_regression.m and logistic_regression.m. For this exercise, fill in the linear_regression_vec.m and logistic_regression_vec.m files with a vectorized implementation of your previous solutions. Uncomment the calling code in ex1a_linreg.m and ex1b_logreg.m and compare the running times of each implementation. Verify that you get similar results to your original solutions!

Debugging: Gradient Checking

So far we have worked with relatively simple algorithms where it is straight-forward to compute the objective function and its gradient with pen-and-paper, and then implement the necessary computations in MATLAB. For more complex models that we will see later (like the back-propagation method for neural networks), the gradient computation can be notoriously difficult to debug and get right. Sometimes a subtly buggy implementation 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. In this section, we describe a method for numerically checking the derivatives computed by your code to make sure that your implementation is correct. Carrying out the derivative checking procedure described here will significantly increase your confidence in the correctness of your code.

Suppose we want to minimize $J(\theta)$ as a function of θ. For this example, suppose $J : \Re \mapsto \Re$, so that $\theta \in \Re$. If we are using minFunc or some other optimization algorithm, then we usually have implemented some function $g(\theta)$ that purportedly computes $\frac{d}{d\theta}J(\theta)$.

How can we check if our implementation of g is correct?

Recall the mathematical definition of the derivative as

$$\frac{d}{d\theta}J(\theta) = \lim_{\epsilon \to 0} \frac{J(\theta + \epsilon) - J(\theta - \epsilon)}{2\epsilon}.$$

Thus, at any specific value of θ, we can numerically approximate the derivative as follows:

$$\frac{J(\theta + \text{EPSILON}) - J(\theta - \text{EPSILON})}{2 \times \text{EPSILON}}$$

In practice, we set EPSILON to a small constant, say around 10^{-4}. (There's a large range of values of EPSILON that should work well, but we don't set EPSILON to be "extremely" small, say 10^{-20}, as that would lead to numerical roundoff errors.)

Thus, given a function $g(\theta)$ that is supposedly computing $\frac{d}{d\theta}J(\theta)$, we can now numerically verify its correctness by checking that

$$g(\theta) \approx \frac{J(\theta + \text{EPSILON}) - J(\theta - \text{EPSILON})}{2 \times \text{EPSILON}}.$$

The degree to which these two values should approximate each other will depend on the details of J. But assuming $\text{EPSILON} = 10^{-4}$, you'll usually find that the left- and right-hand sides of the above will agree to at least 4 significant digits (and often many more).

Now, consider the case where $\theta \in \Re^n$ is a vector rather than a single real number (so that we have n parameters that we want to learn), and $J : \Re^n \mapsto \Re$. We now generalize our derivative checking procedure to the case where θ may be a vector (as in our linear regression and logistic regression examples). If ever we are optimizing over several variables or over matrices, we can always pack these parameters into a long vector and use the same method here to check our derivatives. (This will often need to be done anyway if you want to use off-the-shelf optimization packages.)

Suppose we have a function $g_i(\theta)$ that purportedly computes $\frac{\partial}{\partial\theta_i}J(\theta)$; we'd like to check if g_i is outputting correct derivative values. Let

$$\theta^{(i+)} = \theta + \text{EPSILON} \times \vec{e}_i$$

Where:

$$\vec{e}_i = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

is the i-th basis vector (a vector of the same dimension as θ, with a "1" in the i-th position and "0"s everywhere else). So, $\theta^{(i+)}$ is the same as θ, except its i-th element has been incremented by EPSILON. Similarly, let

$$\theta^{(i-)} = \theta - \text{EPSILON} \times \vec{e}_i$$

be the corresponding vector with the i-th element decreased by EPSILON. We can now numerically verify $g_i(\theta)$'s correctness by checking, for each i, that:

$$g_i(\theta) \approx \frac{J(\theta^{(i+)}) - J(\theta^{(i-)})}{2 \times \text{EPSILON}}.$$

Gradient checker code

As an exercise, try implementing the above method to check the gradient of your linear regression and logistic regression functions. Alternatively, you can use the provided ex1/grad_check.m file (which takes arguments similar to minFunc) and will check $\frac{\partial J(\theta)}{\partial \theta_i}$ for many random choices of *i*.

Softmax Regression

Introduction

Softmax regression (or multinomial logistic regression) is a generalization of logistic regression to the case where we want to handle multiple classes. In logistic regression we assumed that the labels were binary: $y^{(i)} \in \{0, 1\}$. We used such a classifier to distinguish between two kinds of hand-written digits. Softmax regression allows us to handle $y^{(i)} \in \{1, \ldots, K\}$ where *K* is the number of classes.

Recall that in logistic regression, we had a training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ of *m* labeled examples, where the input features are $x^{(i)} \in \Re^{n}$. With logistic regression, we were in the binary classification setting, so the labels were $y^{(i)} \in \{0, 1\}$. Our hypothesis took the form:

$$h_\theta(x) = \frac{1}{1 + \exp(-\theta^\top x)},$$

and the model parameters θ were trained to minimize the cost function

$$J(\theta) = -\left[\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))\right]$$

In the softmax regression setting, we are interested in multi-class classification (as opposed to only binary classification), and so the label *y* can take on *K*different values, rather than only two. Thus, in our training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$, we now have that $y^{(i)} \in \{1, 2, \ldots, K\}$. (Note that our convention will be to index the classes starting from 1, rather than from 0.) For example, in the MNIST digit recognition task, we would have *K* = 10 different classes.

Given a test input *x*, we want our hypothesis to estimate the probability that *P*(*y* = *k* | *x*) for each value of $k = 1, \ldots, K$. I.e., we want to estimate the probability of the class label taking on each of the *K* different possible values. Thus, our hypothesis will output a *K*-dimensional vector (whose elements sum to 1) giving us our *K* estimated probabilities. Concretely, our hypothesis $h_\theta(x)$ takes the form:

$$h_\theta(x) = \begin{bmatrix} P(y=1|x;\theta) \\ P(y=2|x;\theta) \\ \vdots \\ P(y=K|x;\theta) \end{bmatrix} = \frac{1}{\sum_{j=1}^{K} \exp(\theta^{(j)\top} x)} \begin{bmatrix} \exp(\theta^{(1)\top} x) \\ \exp(\theta^{(2)\top} x) \\ \vdots \\ \exp(\theta^{(K)\top} x) \end{bmatrix}$$

Here $\theta^{(1)}, \theta^{(2)}, \ldots, \theta^{(K)} \in \Re^n$ are the parameters of our model. Notice that the term $\frac{1}{\sum_{j=1}^{K} \exp(\theta^{(j)\top} x)}$ normalizes the distribution, so that it sums to one.

For convenience, we will also write θ to denote all the parameters of our model. When you implement softmax regression, it is usually convenient to representθ as a *n*-by-*K* matrix obtained by concatenating $\theta^{(1)}, \theta^{(2)}, \ldots, \theta^{(K)}$ into columns, so that

$$\theta = \left[\begin{array}{cccc} | & | & | & | \\ \theta^{(1)} & \theta^{(2)} & \cdots & \theta^{(K)} \\ | & | & | & | \end{array}\right].$$

Cost Function

We now describe the cost function that we'll use for softmax regression. In the equation below, $1\{\cdot\}$ is the indicator function, so that 1{a true statement} = 1, and 1{a false statement} = 0. For example, 1{2 + 2 = 4} evaluates to 1; whereas 1{1 + 1 = 5} evaluates to 0. Our cost function will be:

$$J(\theta) = -\left[\sum_{i=1}^{m} \sum_{k=1}^{K} 1\left\{y^{(i)} = k\right\} \log \frac{\exp(\theta^{(k)\top} x^{(i)})}{\sum_{j=1}^{K} \exp(\theta^{(j)\top} x^{(i)})} \right]$$

Notice that this generalizes the logistic regression cost function, which could also have been written:

$$\begin{aligned} J(\theta) &= -\left[\sum_{i=1}^{m} (1-y^{(i)}) \log(1-h_\theta(x^{(i)})) + y^{(i)} \log h_\theta(x^{(i)}) \right] \\ &= -\left[\sum_{i=1}^{m} \sum_{k=0}^{1} 1\left\{y^{(i)} = k\right\} \log P(y^{(i)} = k | x^{(i)} ; \theta) \right] \end{aligned}$$

The softmax cost function is similar, except that we now sum over the *K* different possible values of the class label. Note also that in softmax regression, we have that

$$P(y^{(i)} = k | x^{(i)} ; \theta) = \frac{\exp(\theta^{(k)\top} x^{(i)})}{\sum_{j=1}^{K} \exp(\theta^{(j)\top} x^{(i)})}.$$

We cannot solve for the minimum of *J*(θ) analytically, and thus as usual we'll resort to an iterative optimization algorithm. Taking derivatives, one can show that the gradient is:

$$\nabla_{\theta^{(k)}} J(\theta) = -\sum_{i=1}^{m} \left[x^{(i)} \left(1\{ y^{(i)} = k\} - P(y^{(i)} = k | x^{(i)}; \theta) \right) \right]$$

Recall the meaning of the "$\nabla_{\theta^{(k)}}$" notation. In particular, $\nabla_{\theta^{(k)}} J(\theta)$ is itself a vector, so that its *j*-th element is $\frac{\partial J(\theta)}{\partial \theta_{lk}}$ the partial derivative of *J*(θ) with respect to the *j*-th element of $\theta^{(k)}$.

Armed with this formula for the derivative, one can then plug it into a standard optimization package and have it minimize *J*(θ).

Properties of softmax regression parameterization

Softmax regression has an unusual property that it has a "redundant" set of parameters. To explain what this means, suppose we take each of our parameter vectors $\theta^{(j)}$, and subtract some fixed vector ψ from it, so that every $\theta^{(j)}$ is now replaced with $\theta^{(j)} - \psi$ (for every $j = 1, \ldots, k$). Our hypothesis now estimates the class label probabilities as

$$\begin{aligned}
p(y^{(i)} = k | x^{(i)} ; \theta) &= \frac{\exp((\theta^{(k)} - \psi)^\top x^{(i)})}{\sum_{j=1}^K \exp((\theta^{(j)} - \psi)^\top x^{(i)})} \\
&= \frac{\exp(\theta^{(k)\top} x^{(i)}) \exp(-\psi^\top x^{(i)})}{\sum_{j=1}^K \exp(\theta^{(j)\top} x^{(i)}) \exp(-\psi^\top x^{(i)})} \\
&= \frac{\exp(\theta^{(k)\top} x^{(i)})}{\sum_{j=1}^K \exp(\theta^{(j)\top} x^{(i)})}.
\end{aligned}$$

In other words, subtracting ψ from every $\theta^{(j)}$ does not affect our hypothesis' predictions at all! This shows that softmax regression's parameters are "redundant." More formally, we say that our softmax model is overparameterized, meaning that for any hypothesis we might fit to the data, there are multiple parameter settings that give rise to exactly the same hypothesis function h_θ mapping from inputs x to the predictions.

Further, if the cost function $J(\theta)$ is minimized by some setting of the parameters $(\theta^{(1)}, \theta^{(2)}, \ldots, \theta^{(k)})$, then it is also minimized by $(\theta^{(1)} - \psi, \theta^{(2)} - \psi, \ldots, \theta^{(k)} - \psi)$for any value of ψ. Thus, the minimizer of $J(\theta)$ is not unique. (Interestingly, $J(\theta)$ is still convex, and thus gradient descent will not run into local optima problems. But the Hessian is singular/non-invertible, which causes a straightforward implementation of Newton's method to run into numerical problems.)

Notice also that by setting $\psi = \theta^{(K)}$, one can always replace $\theta^{(K)}$ with $\theta^{(K)} - \psi = \vec{0}$ (the vector of all 0's), without affecting the hypothesis. Thus, one could "eliminate" the vector of parameters $\theta^{(K)}$ (or any other $\theta^{(k)}$, for any single value of k), without harming the representational power of our hypothesis. Indeed, rather than optimizing over the $K \cdot n$ parameters $(\theta^{(1)}, \theta^{(2)}, \ldots, \theta^{(K)})$ (where $\theta^{(k)} \in \Re^n$), one can instead set $\theta^{(K)} = \vec{0}$ and optimize only with respect to the $K \cdot n$remaining parameters.

Relationship to Logistic Regression

In the special case where $K = 2$, one can show that softmax regression reduces to logistic regression. This shows that softmax regression is a generalization of logistic regression. Concretely, when $K = 2$, the softmax regression hypothesis outputs

$$h_\theta(x) = \frac{1}{\exp(\theta^{(1)\top}x) + \exp(\theta^{(2)\top}x^{(i)})}\begin{bmatrix}\exp(\theta^{(1)\top}x)\\ \exp(\theta^{(2)\top}x)\end{bmatrix}$$

Taking advantage of the fact that this hypothesis is overparameterized and setting $\psi = \theta^{(2)}$, we can subtract $\theta^{(2)}$ from each of the two parameters, giving us

$$\begin{aligned}
h(x) &= \frac{1}{\exp((\theta^{(1)} - \theta^{(2)})^\top x^{(i)}) + \exp(\vec{0}^\top x)}\left[\exp((\theta^{(1)} - \theta^{(2)})^\top x)\, \exp(\vec{0}^\top x)\right] \\
&= \begin{bmatrix}\frac{1}{1+\exp((\theta^{(1)}-\theta^{(2)})^\top x^{(i)})}\\ \frac{\exp((\theta^{(1)}-\theta^{(2)})^\top x)}{1+\exp((\theta^{(1)}-\theta^{(2)})^\top x^{(i)})}\end{bmatrix} \\
&= \begin{bmatrix}\frac{1}{1+\exp((\theta^{(1)}-\theta^{(2)})^\top x^{(i)})}\\ 1 - \frac{1}{1+\exp((\theta^{(1)}-\theta^{(2)})^\top x^{(i)})}\end{bmatrix}
\end{aligned}$$

Thus, replacing $\theta^{(2)} - \theta^{(1)}$ with a single parameter vector θ', we find that softmax regression predicts the probability of one of the classes as $\frac{1}{1 + \exp(-(\theta')^\top x^{(i)})}$, and that of the other class as $1 - \frac{1}{1 + \exp(-(\theta')^\top x^{(i)})}$, same as logistic regression.

Exercise 1C

Starter code for this exercise is included in the [Starter code GitHub repo] in the ex1/ directory.

In this exercise you will train a classifier to handle all 10 digits in the MNIST dataset. The code is very similar to that used for Exercise 1B except that it will load the entire MNIST train and test sets (instead of just the 0 and 1 digits), and the labels $y^{(i)}$ have 1 added to them so that $y^{(i)} \in \{1, \ldots, 10\}$. (The change in the labels allows you to use $y^{(i)}$ as an index into a matrix.)

The code performs the same operations as in Exercise 1B: it loads the train and test data, adding an intercept term, then calls minFunc with the softmax_regression_vec.m file as the objective function. When training is complete, it will print out training and testing accuracies for the 10-class digit recognition problem.

Your task is to implement the softmax_regression_vec.m file to compute the softmax objective function $J(\theta;X,y)$ and store it in the variable *f*. You must also compute the gradient $\nabla_\theta J(\theta; X, y)$ and store it in the variable *g*. Don't forget that minFunc supplies the parameters θ as a vector. The starter code will reshapeθ into a n-by-(K-1) matrix (for K=10 classes). You also need to remember to reshape the returned gradient *g* back into a vector using *g* = *g*(:);

You can start out with a for-loop version of the code if necessary to get the gradient right. (Be sure to use the gradient check debugging strategy covered earlier!) However, you might find that this implementation is too slow to run the optimizer all the way through. After you get the gradient right with a slow version of the code, try to vectorize your code as well as possible before running the full experiment.

Here are a few MATLAB tips that you might find useful for implementing or speeding up your code (though these may or may not be useful depending on your implementation strategy):

1. Suppose we have a matrix *A* and we want to extract a single element from each row, where the column of the element to be extracted from row *i* is stored in$y(i)$, where *y* is a row vector. We can use the sub2ind() function like this:

```
I=sub2ind(size(A), 1:size(A,1), y);
values = A(I);
```

This code will take each pair of indices (*i*,*j*) where *i* comes from the second argument and *j* comes from the corresponding element of the third argument, and compute the corresponding 1D index into *A* for the (*i*,*j*)'th element. So, *I*(1) will be the index for the element at location (1,*y*(1)), and *I*(2) will be the index for the element at (2,*y*(2)).

2. When you compute the predicted label probabilities

$$\hat{y}_k^{(i)} = \exp(\theta_{:,k}^\top x^{(i)}) / (\sum_{j=1}^{K} \exp(\theta_{:,j}^\top x^{(i)}))$$

try to use matrix multiplications and bsxfun to speed up the computation. For example, once θ is in matrix form, you can compute the products for every example and the first 9 classes using $a = \theta^\top X$. (Recall that the 10th class is left out of θ, so that *a*(10,:) is just assumed to be 0.)

Debugging: Bias and Variance

Thus far, we have seen how to implement several types of machine learning algorithms. Our usual goal is to achieve the highest possible prediction accuracy on novel test data that our algorithm did not see during training. It turns out that the our accuracy on the *training* data is an upper bound on the accuracy we can expect to achieve on the testing data. (We can sometimes get lucky and do better on a small sample of test data; but on average we will tend to do worse.) In some sense, the training data is "easier" because the algorithm has been trained for those examples specifically and thus there is a gap between the training and testing accuracy.

Debugging: Optimizers and objectives

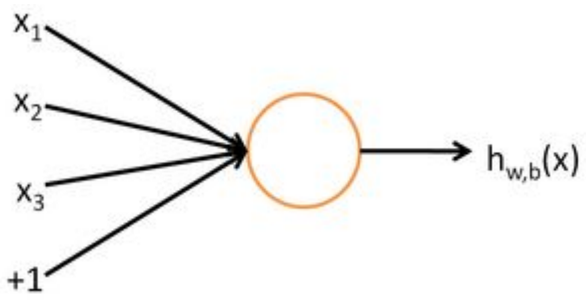
There is currently no text in this page. You can search for this page title in other pages, or search the related logs.

Supervised Learning and Optimization

Multi-Layer 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 will begin by describing the simplest possible neural network, one which comprises a single "neuron." We will 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

$$h_{W,b}(x) = f(W^Tx) = f(\textstyle\sum_{i=1}^{3} W_i x_i + b)$$

where $f : \Re \mapsto \Re$ is called the **activation function**. In these notes, we will choose $f(\cdot)$ to be the sigmoid function:

$$f(z) = \frac{1}{1+\exp(-z)}.$$

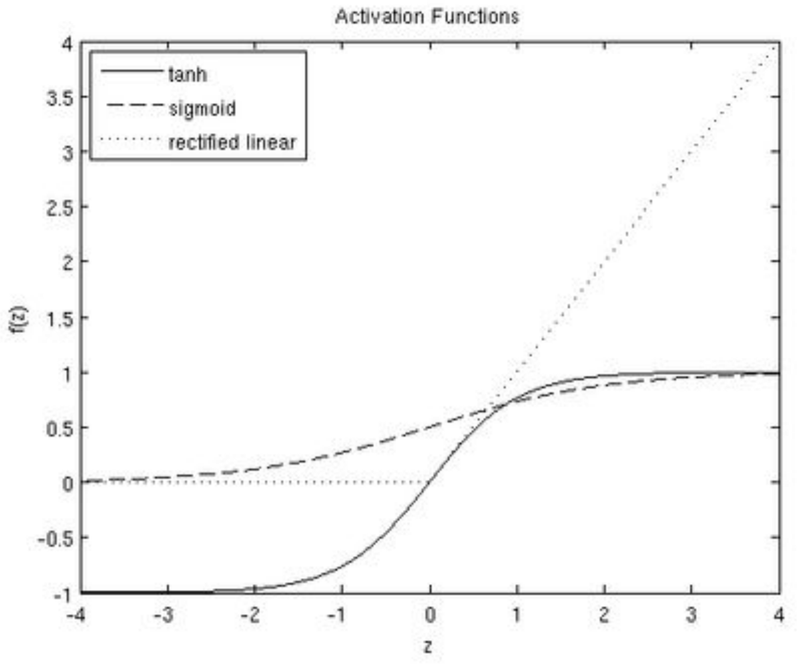
Thus, our single neuron corresponds exactly to the input-output mapping defined by logistic regression.

Although these notes will use the sigmoid function, it is worth noting that another common choice for f is the hyperbolic tangent, or tanh, function:

$$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}.$$

Recent research has found a different activation function, the rectified linear function, often works better in practice for deep neural networks. This activation function is different from sigmoid and tanhbecause it is not bounded or continuously differentiable. The rectified linear activation function is given by,$f(z) = \max(0,x)$.

Here are plots of the sigmoid, tanh and rectified linear functions:



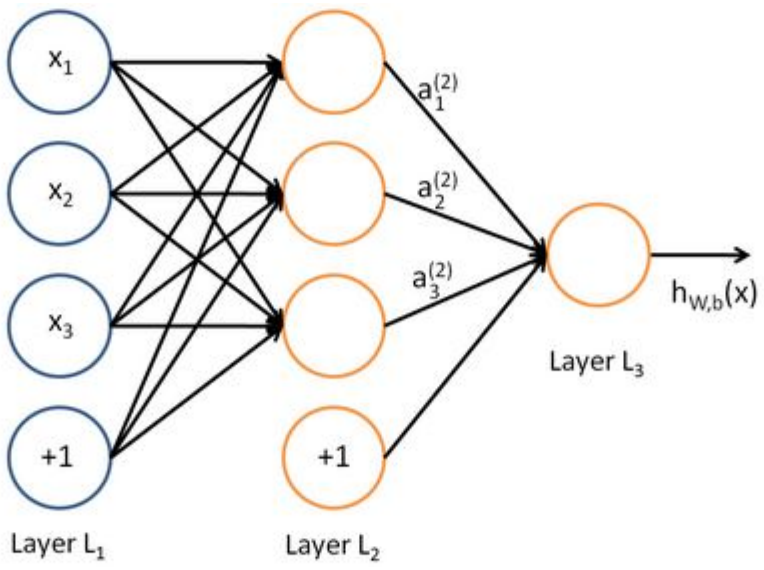
The tanh(z) function is a rescaled version of the sigmoid, and its output range is [− 1,1] instead of [0,1]. The rectified linear function is piece-wise linear and saturates at exactly 0 whenever the input z is less than 0.

Note that unlike some other venues (including the OpenClassroom videos, and parts of CS229), we are not using the convention here of $x_0 = 1$. Instead, the intercept term is handled separately by the parameter b.

Finally, one identity that'll be useful later: If $f(z) = 1 / (1 + \exp(- z))$ is the sigmoid function, then its derivative is given by $f'(z) = f(z)(1 - f(z))$. (If f is the tanh function, then its derivative is given by $f'(z) = 1 - (f(z))^2$.) You can derive this yourself using the definition of the sigmoid (or tanh) function. The rectified linear function has gradient 0 when $z \leq 0$ and 1 otherwise. The gradient is undefined at $z = 0$, though this doesn't cause problems in practice because we average the gradient over many training examples during optimization.

Neural Network model

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 $W^{(1)} \in \Re^{3\times 3}$, and $W^{(2)} \in \Re^{1\times 3}$. 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:

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

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., $f([z_1,z_2,z_3]) = [f(z_1),f(z_2),f(z_3)]$), then we can write the equations above more compactly as:

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

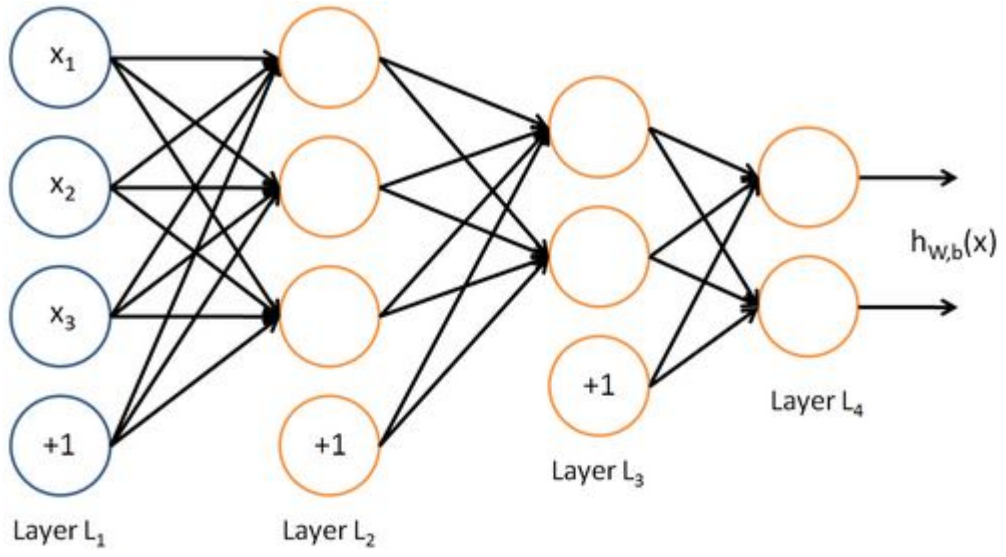
We call this step **forward propagation.** More generally, recalling that we also use $a^{(1)} = x$ to also denote the values from the input layer, then given layer *l*'s activations $a^{(l)}$, we can compute layer $l + 1$'s activations $a^{(l+1)}$ as:

$$\begin{aligned}
z^{(l+1)} &= W^{(l)}a^{(l)} + b^{(l)} \\
a^{(l+1)} &= f(z^{(l+1)})
\end{aligned}$$

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 the equations above that describe the forward propagation step. 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 $y^{(i)} \in \Re^2$. 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.)

Backpropagation Algorithm

Suppose we have a fixed training set $\{(x^{(1)},y^{(1)}),\ldots,(x^{(m)},y^{(m)})\}$ of m training examples. We can train our neural network using batch gradient descent. In detail, for a single training example (x,y), we define the cost function with respect to that single example to be:

$$J(W,b;x,y) = \frac{1}{2}\left\|h_{W,b}(x) - y\right\|^2.$$

This is a (one-half) squared-error cost function. Given a training set of m examples, we then define the overall cost function to be:

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

The first term in the definition of $J(W,b)$ is an average sum-of-squares error term. The second term is a regularization term (also called a **weight decay** term) that tends to decrease the magnitude of the weights, and helps prevent overfitting.

[Note: Usually weight decay is not applied to the bias terms $b_i^{(l)}$, as reflected in our definition for $J(W,b)$. Applying weight decay to the bias units usually makes only a small difference to the final network, however. If you've taken CS229 (Machine Learning) at Stanford or watched the course's videos on YouTube, you may also recognize this weight decay 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.]

The **weight decay parameter** λ controls the relative importance of the two terms. Note also the slightly overloaded notation: $J(W,b;x,y)$ is the squared error cost with respect to a single example; $J(W,b)$is the overall cost function, which includes the weight decay term.

This cost function above is often used both for classification and for regression problems. For classification, we let $y = 0$ or 1 represent the two class labels (recall that the sigmoid activation function outputs values in [0,1]; if we were using a tanh activation function, we would instead use -1 and +1 to denote the labels). For regression problems, we first scale our outputs to ensure that they lie in the [0,1] range (or if we were using a tanh activation function, then the $[-1,1]$ range).

Our goal is to minimize $J(W,b)$ 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 $Normal(0, \epsilon^2)$ distribution for some small ϵ, say 0.01), and then apply an optimization algorithm such as batch gradient descent.
Since $J(W,b)$ is a non-convex function, gradient descent is susceptible to local optima; however, in practice gradient descent usually works fairly well. 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, $W^{(1)}_{ij}$ will be the same for all values of i, so that $a^{(2)}_1 = a^{(2)}_2 = a^{(2)}_3 = \ldots$ for any input x). The random initialization serves the purpose of **symmetry breaking**.

One iteration of gradient descent updates the parameters W,b as follows:

$$
\begin{aligned}
W_{ij}^{(l)} &= W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(W,b) \\
b_{i}^{(l)} &= b_{i}^{(l)} - \alpha \frac{\partial}{\partial b_{i}^{(l)}} J(W,b)
\end{aligned}
$$

where α is the learning rate. The key step is computing the partial derivatives above. We will now describe the **backpropagation** algorithm, which gives an efficient way to compute these partial derivatives.

We will first describe how backpropagation can be used to compute $\frac{\partial}{\partial W_{ij}^{(l)}} J(W,b;x,y)$ and $\frac{\partial}{\partial b_{i}^{(l)}} J(W,b;x,y)$, the partial derivatives of the cost function $J(W,b;x,y)$ defined with respect to a single example (x,y). Once we can compute these, we see that the derivative of the overall cost function $J(W,b)$ can be computed as:

$$
\begin{aligned}
\frac{\partial}{\partial W_{ij}^{(l)}} J(W,b) &= \left[\frac{1}{m} \sum_{i=1}^m \frac{\partial}{\partial W_{ij}^{(l)}} J(W,b; x^{(i)}, y^{(i)}) \right] + \lambda W_{ij}^{(l)} \\
\frac{\partial}{\partial b_{i}^{(l)}} J(W,b) &= \frac{1}{m}\sum_{i=1}^m \frac{\partial}{\partial b_{i}^{(l)}} J(W,b; x^{(i)}, y^{(i)})
\end{aligned}
$$

The two lines above differ slightly because weight decay is applied to W but not b.

The intuition behind the backpropagation 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 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. Compute the desired partial derivatives, which are given as:
$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W,b;x,y) = a_j^{(l)} \delta_i^{(l+1)}$$
$$\frac{\partial}{\partial b_i^{(l)}} J(W,b;x,y) = \delta_i^{(l+1)}.$$

Finally, we can also re-write the algorithm using matrix-vectorial notation. We will use "●" 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 $f'([z_1, z_2, z_3]) = [f'(z_1), f'(z_2), f'(z_3)]$).

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 the equations defining the forward propagation steps
2. For the output layer (layer n_l), set

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

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. Compute the desired partial derivatives:

$$\begin{aligned} \nabla_{W^{(l)}} J(W,b;x,y) &= \delta^{(l+1)} (a^{(l)})^T, \\ \nabla_{b^{(l)}} J(W,b;x,y) &= \delta^{(l+1)}. \end{aligned}$$

Implementation note: In steps 2 and 3 above, we need to compute $f'(z_i^{(l)})$ for each value of i. Assuming $f(z)$ is the sigmoid 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

$$f'(z_i^{(l)}) = a_i^{(l)} (1 - a_i^{(l)})$$

Finally, we are ready to describe the full gradient descent algorithm. In the pseudo-code below, $\Delta W^{(l)}$ is a matrix (of the same dimension as $W^{(l)}$), and $\Delta b^{(l)}$ is a vector (of the same dimension as $b^{(l)}$). Note that in this notation, "$\Delta W^{(l)}$" is a matrix, and in particular it isn't "Δ times $W^{(l)}$." We implement one iteration of batch gradient descent as follows:

1. Set $\Delta W^{(l)} := 0, \Delta b^{(l)} := 0$ (matrix/vector of zeros) for all l.

2. For $i = 1$ to m,

    a. Use backpropagation to compute $\nabla_{W^{(l)}} J(W,b;x,y)$ and $\nabla_{b^{(l)}} J(W,b;x,y)$.

    b. Set

    $$\Delta W^{(l)} := \Delta W^{(l)} + \nabla_{W^{(l)}} J(W,b;x,y).$$

    c. Set

    $$\Delta b^{(l)} := \Delta b^{(l)} + \nabla_{b^{(l)}} J(W,b;x,y).$$

3. Update the parameters:

$$W^{(l)} = W^{(l)} - \alpha \left[\left(\frac{1}{m} \Delta W^{(l)} \right) + \lambda W^{(l)} \right]$$

$$b^{(l)} = b^{(l)} - \alpha \left[\frac{1}{m} \Delta b^{(l)} \right]$$

To train our neural network, we can now repeatedly take steps of gradient descent to reduce our cost function $J(W,b)$.

Template:Sparse Autoencoder

Template:Languages

Exercise: Supervised Neural Network

In this exercise, you will train a neural network classifier to classify the 10 digits in the MNIST dataset. The output unit of your neural network is identical to the softmax regression function you created in the Softmax Regression exercise. The softmax regression function alone did not fit the training set well, an example of *underfitting*. In comparison, a neural network has lower bias and should better fit the training set. In the section on Multi-Layer Neural Networks we covered the backpropagation algorithm to compute gradients for all parameters in the network using the squared error loss function. For this exercise, we need the same cost function as used for softmax regression (cross entropy), instead of the squared error function.

The cost function is nearly identical to the softmax regression cost function. Note that instead of making predictions from the input data *x* the softmax function takes as input the final hidden layer of the network $h_{W,b}(x)$. The loss function is thus,

$$J(\theta) = -\left[\sum_{i=1}^{m}\sum_{k=1}^{K} 1\left\{y^{(i)} = k\right\} \log \frac{\exp(\theta^{(k)\top} h_{W,b}(x^{(i)}))}{\sum_{j=1}^{K}\exp(\theta^{(j)\top} h_{W,b}(x)^{(i)}))}\right].$$

The difference in cost function results in a different value for the error term at the output layer ($\delta^{(n_l)}$). For the cross entropy cost we have,

$$\delta^{(n_l)} = -\sum_{i=1}^{m}\left[\left(1\{y^{(i)} = k\} - P(y^{(i)} = k|x^{(i)};\theta)\right)\right]$$

Using this term, you should be able to derive the full backpropagation algorithm to compute gradients for all network parameters.

Using the starter code given, create a cost function which does forward propagation of your neural network, and computes gradients. As before, we will use the minFunc optimization package to do gradient-based optimization. Remember to numerically check your gradient computations! Your implementation should support training neural networks with multiple hidden layers. As you develop your code, follow this path of milestones:

▪ Implement and gradient check a single hidden layer network. When performing the gradient check, you may want to reduce the input dimensionality and number of examples by cropping the training data matrix. Similarly, when gradient checking you should use a small number of hidden units to reduce computation time.

▪ Gradient check your implementation with a two hidden layer network.

▪ Train and test various network architectures. You should be able to achieve 100% training set accuracy with a single hidden layer of 256 hidden units. Because the network has many parameters, there is a danger of overfitting. Experiment with layer size, number of hidden layers, and weight decay penalty to understand what types of architectures perform best. Can you find a network with multiple hidden layers which outperforms your best single hidden layer architecture?

▪ (Optional) Extend your code to support multiple choices for hidden unit nonlinearity (sigmoid, tanh, and rectified linear).

Supervised Convolutional Neural Network

Feature Extraction Using Convolution

Overview

In the previous exercises, you worked through problems which involved images that were relatively low in resolution, such as small image patches and small images of hand-written digits. In this section, we will develop methods which will allow us to scale up these methods to more realistic datasets that have larger images.

Fully Connected Networks

In the sparse autoencoder, one design choice that we had made was to "fully connect" all the hidden units to all the input units. On the relatively small images that we were working with (e.g., 8x8 patches for the sparse autoencoder assignment, 28x28 images for the MNIST dataset), it was computationally feasible to learn features on the entire image. However, with larger images (e.g., 96x96 images) learning features that span the entire image (fully connected networks) is very computationally expensive--you would have about 10^4 input units, and assuming you want to learn 100 features, you would have on the order of 10^6 parameters to learn. The feedforward and backpropagation computations would also be about 10^2 times slower, compared to 28x28 images.

Locally Connected Networks

One simple solution to this problem is to restrict the connections between the hidden units and the input units, allowing each hidden unit to connect to only a small subset of the input units. Specifically, each hidden unit will connect to only a small contiguous region of pixels in the input. (For input modalities different than images, there is often also a natural way to select "contiguous groups" of input units to connect to a single hidden unit as well; for example, for audio, a hidden unit might be connected to only the input units corresponding to a certain time span of the input audio clip.)

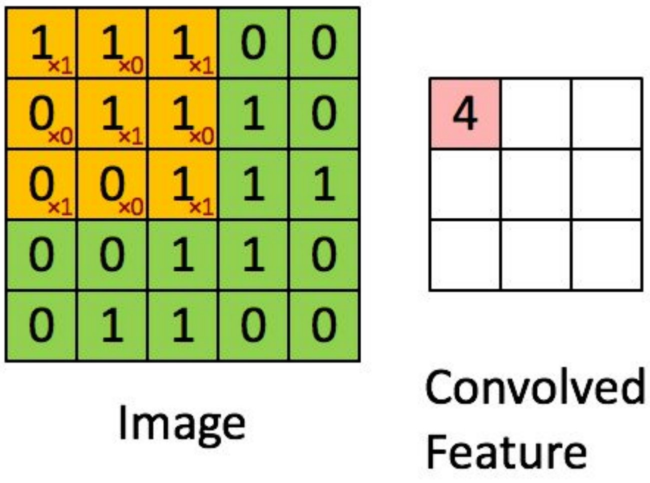
This idea of having locally connected networks also draws inspiration from how the early visual system is wired up in biology. Specifically, neurons in the visual cortex have localized receptive fields (i.e., they respond only to stimuli in a certain location).

Convolutions

Natural images have the property of being **stationary**, meaning that the statistics of one part of the image are the same as any other part. This suggests that the features that we learn at one part of the image can also be applied to other parts of the image, and we can use the same features at all locations.

More precisely, having learned features over small (say 8x8) patches sampled randomly from the larger image, we can then apply this learned 8x8 feature detector anywhere in the image. Specifically, we can take the learned 8x8 features and **convolve** them with the larger image, thus obtaining a different feature activation value at each location in the image.

To give a concrete example, suppose you have learned features on 8x8 patches sampled from a 96x96 image. Suppose further this was done with an autoencoder that has 100 hidden units. To get the convolved features, for every 8x8 region of the 96x96 image, that is, the 8x8 regions starting at $(1,1),(1,2),\ldots(89,89)$, you would extract the 8x8 patch, and run it through your trained sparse autoencoder to get the feature activations. This would result in 100 sets 89x89 convolved features.



Formally, given some large $r \times c$ images x_{large}, we first train a sparse autoencoder on small $a \times b$ patches x_{small} sampled from these images, learning k features $f = \sigma(W^{(1)}x_{small} + b^{(1)})$ (where σ is the sigmoid function), given by the weights $W^{(1)}$ and biases $b^{(1)}$ from the visible units to the hidden units. For every $a \times b$ patch x_s in the large image, we compute $f_s = \sigma(W^{(1)}x_s + b^{(1)})$, giving us$f_{convolved}$, a

$$k \times (r - a + 1) \times (c - b + 1)$$

array of convolved features.

In the next section, we further describe how to "pool" these features together to get even better features for classification.

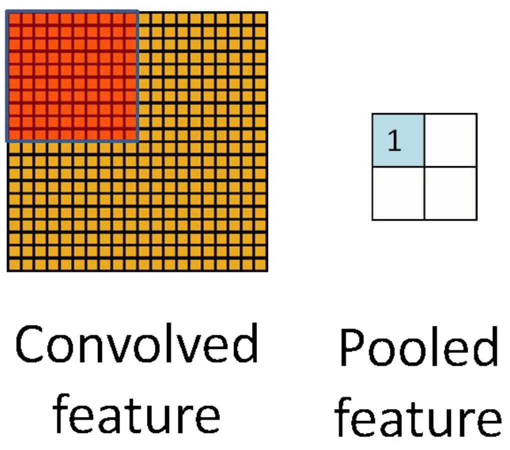
Pooling

Pooling: Overview

After obtaining features using convolution, we would next like to use them for classification. In theory, one could use all the extracted features with a classifier such as a softmax classifier, but this can be computationally challenging. Consider for instance images of size 96x96 pixels, and suppose we have learned 400 features over 8x8 inputs. Each convolution results in an output of size $(96 - 8 + 1) * (96 - 8 + 1) = 7921$, and since we have 400 features, this results in a vector of $89^2 * 400 = 3{,}168{,}400$ features per example. Learning a classifier with inputs having 3+ million features can be unwieldy, and can also be prone to over-fitting.

To address this, first recall that we decided to obtain convolved features because images have the "stationarity" property, which implies that features that are useful in one region are also likely to be useful for other regions. Thus, to describe a large image, one natural approach is to aggregate statistics of these features at various locations. For example, one could compute the mean (or max) value of a particular feature over a region of the image. These summary statistics are much lower in dimension (compared to using all of the extracted features) and can also improve results (less over-fitting). We aggregation operation is called this operation **pooling**, or sometimes **mean pooling** or **max pooling** (depending on the pooling operation applied).

The following image shows how pooling is done over 4 non-overlapping regions of the image.



Pooling for Invariance

If one chooses the pooling regions to be contiguous areas in the image and only pools features generated from the same (replicated) hidden units. Then, these pooling units will then be **translation invariant**. This means that the same (pooled) feature will be active even when the image undergoes (small) translations. Translation-invariant features are often desirable; in many tasks (e.g., object detection, audio recognition), the label of the example (image) is the same even when the image is translated. For example, if you were to take an MNIST digit and translate it left or right, you would want your classifier to still accurately classify it as the same digit regardless of its final position.

Formal description

Formally, after obtaining our convolved features as described earlier, we decide the size of the region, say $m \times n$ to pool our convolved features over. Then, we divide our convolved features into disjoint $m \times n$ regions, and take the mean (or maximum) feature activation over these regions to obtain the pooled convolved features. These pooled features can then be used for classification.

Exercise: Convolution and Pooling

Convolution and Pooling

In this exercise you will and test convolution and pooling functions. We have provided some starter code. You should write your code at the places indicated

"YOUR CODE HERE" in the files. For this exercise, you will need to modify **cnnConvolve.m** and **cnnPool.m**.

Dependencies

The following additional files are required for this exercise:MNIST helper functions

Starter Code

Step 1: Implement and test convolution

In this step, you will implement the convolution function, and test it on a small part of the data set to ensure that you have implemented it correctly.

Step 1a: Implement convolution

Implement convolution, as described in Feature Extraction Using Convolution, in the function cnnConvolve in cnnConvolve.m. Implementing convolution is somewhat involved, so we will guide you through the process below.

First, we want to compute $\sigma(Wx_{(r,c)} + b)$ for all valid (r,c) (valid meaning that the entire 8x8 patch is contained within the image; this is as opposed to a full convolution, which allows the patch to extend outside the image, with the area outside the image assumed to be 0), where W and b are the learned weights and biases from the input layer to the hidden layer, and $x_{(r,c)}$ is the 8x8 patch with the upper left corner at (r,c). To accomplish this, one naive method is to loop over all such patches and compute $\sigma(Wx_{(r,c)} + b)$ for each of them; while this is fine in theory, it can very slow. Hence, we usually use MATLAB's built in convolution functions, which are well optimized.

Observe that the convolution above can be broken down into the following three small steps. First, compute $Wx_{(r,c)}$ for all (r,c). Next, add b to all the computed values. Finally, apply the sigmoid function to the resulting values. This doesn't seem to buy you anything, since the first step still requires a loop. However, you can replace the loop in the first step with one of MATLAB's optimized convolution functions, conv2, speeding up the process significantly.

However, there are two important points to note in using conv2. First, conv2 performs a 2-D convolution, but you have 4 "dimensions" - image number, filter (or feature) number, row of image and column of image - that you want to convolve over. Because of this, you will have to convolve each filter separately for each image, using the row and column of the image as the 2 dimensions you convolve over. This means that you will need two outer loops over the image number imageNum and filter number filterNum. Inside the two nested for-loops, you will perform a conv2 2-D convolution, using the weight matrix for the filterNum-th filter and the image matrix for the imageNum-th image.

Second, because of the mathematical definition of convolution, the filter matrix must be "flipped" before passing it to conv2. The following implementation tip explains the "flipping" of feature matrices when using MATLAB's convolution functions:

Implementation tip: Using conv2 and convn

Because the mathematical definition of convolution involves "flipping" the matrix to convolve with (reversing its rows and its columns), to use MATLAB's convolution functions, you must first "flip" the weight matrix so that when MATLAB "flips" it according to the mathematical definition the entries will be at the correct place. For example, suppose you wanted to convolve two matrices image (a large image) and W (the feature) using conv2(image, W), and W is a 3x3 matrix as below:

$$W = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

If you use conv2(image, W), MATLAB will first "flip" W, reversing its rows and columns, before convolving W with image, as below:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \xrightarrow{flip} \begin{pmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{pmatrix}$$

If the original layout of W was correct, after flipping, it would be incorrect. For the layout to be correct after flipping, you will have to flip W before passing it into conv2, so that after MATLAB flips W inconv2, the layout will be correct. For conv2, this means reversing the rows and columns, which can be done by rotating W 90 degrees twice with rot90 as shown below:

```
% Flip W for use in conv2
W = rot90(W,2);
```

Next, to each of the convolvedFeatures, you should then add *b*, the corresponding bias for the filterNum-th filter.

Step 1b: Check your convolution

We have provided some code for you to check that you have done the convolution correctly. The code randomly checks the convolved values for a number of (feature, row, column) tuples by computing the feature activations using randomly generated features and images from the MNIST dataset.

Step 2: Implement and test pooling

Step 2a: Implement pooling

Implement pooling in the function cnnPool in cnnPool.m. You should implement mean pooling (i.e., averaging over feature responses) for this part. This can be done efficiently using the conv2 function as well. The inputs are the responses of each image with each filter computed in the previous step. Convolve each of these with a matrix of ones followed by a subsampling and averaging. Make sure to use the "valid" border handling convolution.

Step 2b: Check your pooling

We have provided some code for you to check that you have done the pooling correctly. The code runs cnnPool against a test matrix to see if it produces the expected result.

Optimization: Stochastic Gradient Descent

Overview

Batch methods, such as limited memory BFGS, which use the full training set to compute the next update to parameters at each iteration tend to converge very well to local optima. They are also straight forward to get working provided a good off the shelf implementation (e.g. minFunc) because they have very few hyper-parameters to tune. However, often in practice computing the cost and gradient for the entire training set can be very slow and sometimes intractable on a single machine if the dataset is too big to fit in main memory. Another issue with batch optimization methods is that they don't give an easy way to incorporate new data in an 'online' setting. Stochastic Gradient Descent (SGD) addresses both of these issues by following the negative gradient of the objective after seeing only a single or a few training examples. The use of SGD In the neural network setting is motivated by the high cost of running back propagation over the full training set. SGD can overcome this cost and still lead to fast convergence.

Stochastic Gradient Descent

The standard gradient descent algorithm updates the parameters θ of the objective $J(\theta)$ as,

$$\theta = \theta - \alpha \nabla_\theta E[J(\theta)]$$

where the expectation in the above equation is approximated by evaluating the cost and gradient over the full training set. Stochastic Gradient Descent (SGD) simply does away with the expectation in the update and computes the gradient of the parameters using only a single or a few training examples. The new update is given by,

$$\theta = \theta - \alpha \nabla_\theta J(\theta; x^{(i)}, y^{(i)})$$

with a pair ($x^{(i)}$,$y^{(i)}$) from the training set.

Generally each parameter update in SGD is computed w.r.t a few training examples or a minibatch as opposed to a single example. The reason for this is twofold: first this reduces the variance in the parameter update and can lead to more stable convergence, second this allows the computation to take advantage of highly optimized matrix operations that should be used in a well vectorized computation of the cost and gradient. A typical minibatch size is 256, although the optimal size of the minibatch can vary for different applications and architectures.

In SGD the learning rate α is typically much smaller than a corresponding learning rate in batch gradient descent because there is much more variance in the update. Choosing the proper learning rate and schedule (i.e. changing the value of the learning rate as learning progresses) can be fairly difficult. One standard method that works well in practice is to use a small enough constant learning rate that gives stable convergence in the initial epoch (full pass through the training set) or two of training and then halve the value of the learning rate as convergence slows down. An even better approach is to evaluate a held out set after each epoch and anneal the learning rate when the change in objective between epochs is below a small threshold. This tends to give good convergence to a local optima. Another commonly used schedule is to anneal the learning rate at each iteration *t* as where *a* and *b* dictate the initial learning rate and when the annealing begins respectively. More sophisticated methods include using a backtracking line search to find the optimal update.

One final but important point regarding SGD is the order in which we present the data to the algorithm. If the data is given in some meaningful order, this can bias the gradient and lead to poor convergence. Generally a good method to avoid this is to randomly shuffle the data prior to each epoch of training.

Momentum

If the objective has the form of a long shallow ravine leading to the optimum and steep walls on the sides, standard SGD will tend to oscillate across the narrow ravine since the negative gradient will point down one of the steep sides rather than along the ravine towards the optimum. The objectives of deep architectures have this form near local optima and thus standard SGD can lead to very slow convergence particularly after the initial steep gains. Momentum is one method for pushing the objective more quickly along the shallow ravine. The momentum update is given by,

$$\begin{aligned} v &= \gamma v + \alpha \nabla_\theta J(\theta; x^{(i)}, y^{(i)}) \\ \theta &= \theta - v \end{aligned}$$

In the above equation *v* is the current velocity vector which is of the same dimension as the parameter vector θ. The learning rate α is as described above, although when using momentum α may need to be smaller since the magnitude of the gradient will be larger. Finally $\gamma \in (0, 1]$ determines for how many iterations the previous gradients are incorporated into the current update. Generally γ is set to 0.5 until the initial learning stabilizes and then is increased to 0.9 or higher.

Convolutional Neural Network

Overview

A Convolutional Neural Network (CNN) is comprised of one or more convolutional layers (often with a subsampling step) and then followed by one or more fully connected layers as in a standardmultilayer neural network. The architecture of a CNN is designed to take advantage of the 2D structure of an input image (or other 2D input such as a speech signal). This is achieved with local connections and tied weights followed by some form of pooling which results in translation invariant features. Another benefit of CNNs is that they are easier to train and have many fewer parameters than fully connected networks with the same number of hidden units. In this article we will discuss the architecture of a CNN and the back propagation algorithm to compute the gradient with respect to the parameters of the model in order to use gradient based optimization. See the respective tutorials on convolution and pooling for more details on those specific operations.

Architecture

A CNN consists of a number of convolutional and subsampling layers optionally followed by fully connected layers. The input to a convolutional layer is a m x m x r image where m is the height and width of the image and r is the number of channels, e.g. an RGB image has $r = 3$. The convolutional layer will have k filters (or kernels) of size n x n x q where n is smaller than the dimension of the image and $q$can either be the same as the number of channels r or smaller and may vary for each kernel. The size of the filters gives rise to the locally connected structure which are each convolved with the image to produce k feature maps of size $m - n + 1$. Each map is then subsampled typically with mean or max pooling over p x p contiguous regions where p ranges between 2 for small images (e.g. MNIST) and is usually not more than 5 for larger inputs. Either before or after the subsampling layer an additive bias and sigmoidal nonlinearity is applied to each feature map. The figure below illustrates a full layer in a CNN consisting of convolutional and subsampling sublayers. Units of the same color have tied weights.

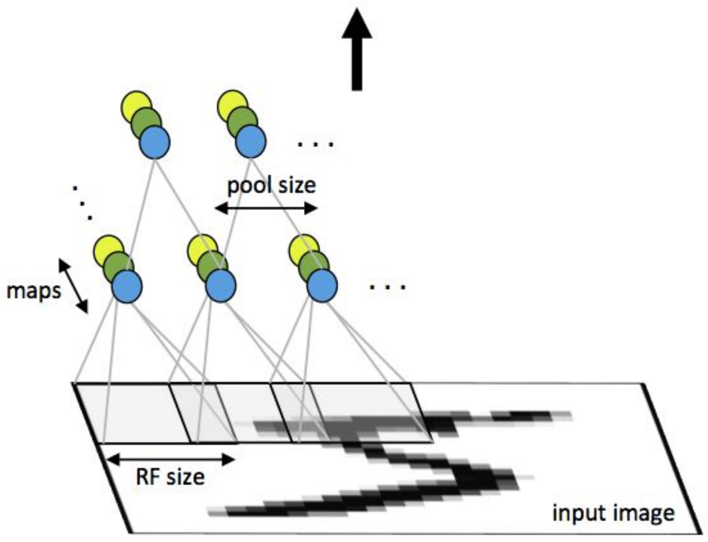


Fig 1: First layer of a convolutional neural network with pooling. Units of the same color have tied weights and units of different color represent different filter maps.

After the convolutional layers there may be any number of fully connected layers. The densely connected layers are identical to the layers in a standard multilayer neural network.

Back Propagation

Let $\delta^{(l+1)}$ be the error term for the $(l + 1)$-st layer in the network with a cost function $J(W,b;x,y)$ where (W,b) are the parameters and (x,y) are the training data and label pairs. If the l-th layer is densely connected to the $(l + 1)$-st layer, then the error for the l-th layer is computed as

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

and the gradients are

$$\nabla_{W^{(l)}} J(W, b; x, y) = \delta^{(l+1)} (a^{(l)})^T,$$
$$\nabla_{b^{(l)}} J(W, b; x, y) = \delta^{(l+1)}.$$

If the *l*-th layer is a convolutional and subsampling layer then the error is propagated through as

$$\delta_k^{(l)} = \text{upsample}\left((W_k^{(l)})^T \delta_k^{(l+1)}\right) \bullet f'(z_k^{(l)})$$

Where *k* indexes the filter number and $f'(z_k^{(l)})$ is the derivative of the activation function. The upsample operation has to propagate the error through the pooling layer by calculating the error w.r.t to each unit incoming to the pooling layer. For example, if we have mean pooling then upsample simply uniformly distributes the error for a single pooling unit among the units which feed into it in the previous layer. In max pooling the unit which was chosen as the max receives all the error since very small changes in input would perturb the result only through that unit.

Finally, to calculate the gradient w.r.t to the filter maps, we rely on the border handling convolution operation again and flip the error matrix $\delta_k^{(l)}$ the same way we flip the filters in the convolutional layer.

$$\nabla_{W_k^{(l)}} J(W, b; x, y) = \sum_{i=1}^{m} (a_i^{(l)}) * \texttt{rot90}(\delta_k^{(l+1)}, 2),$$
$$\nabla_{b_k^{(l)}} J(W, b; x, y) = \sum_{a,b} (\delta_k^{(l+1)})_{a,b}.$$

Where $a^{(l)}$ is the input to the *l*-th layer, and $a^{(1)}$ is the input image. The operation $(a_i^{(l)}) * \delta_k^{(l+1)}$ is the "valid" convolution between *i*-th input in the *l*-th layer and the error w.r.t. the *k*-th filter.

Exercise: Convolutional Neural Network

Overview

In this exercise you will implement a convolutional neural network for digit classification. The architecture of the network will be a convolution and subsampling layer followed by a densely connected output layer which will feed into the softmax regression and cross entropy objective. You will use mean pooling for the subsampling layer. You will use the back-propagation algorithm to calculate the gradient with respect to the parameters of the model. Finally you will train the parameters of the network with stochastic gradient descent and momentum.

We have provided some MATLAB starter code. You should write your code at the places indicated in the files "YOUR CODE HERE". You have to complete the following files: **cnnCost.m**,**minFuncSGD.m**. The starter code in **cnnTrain.m** shows how these functions are used.

Dependencies

Convolutional Network starter code

MNIST helper functions

We strongly suggest that you complete the convolution and pooling, multilayer supervised neural network and softmax regression exercises prior to starting this one.

Implement Convolutional Neural Network

Step 0: Initialize Parameters and Load Data

In this step we initialize the parameters of the convolutional neural network. You will be using 10 filters of dimension 9x9, and a non-overlapping, contiguous 2x2 pooling region.

We also load the MNIST training data here as well.

Step 1: Implement CNN Objective

Implement the CNN cost and gradient computation in this step. Your network will have two layers. The first layer is a convolutional layer followed by mean pooling and the second layer is a densely connected layer into softmax regression. The cost of the network will be the standard cross entropy between the predicted probability distribution over 10 digit classes for each image and the ground truth distribution.

Step 1a: Forward Propagation

Convolve every image with every filter, then mean pool the responses. This should be similar to the implementation from the convolution and pooling exercise using MATLAB's conv2 function. You will need to store the activations after the convolution but before the pooling for efficient back propagation later.

Following the convolutional layer, we unroll the subsampled filter responses into a 2D matrix with each column representing an image. Using the activationsPooled matrix, implement a standard softmax layer following the style of the softmax regression exercise.

Step 1b: Calculate Cost

Generate the ground truth distribution using MATLAB's sparse function from the labels given for each image. Using the ground truth distribution, calculate the cross entropy cost between that and the predicted distribution.

Note at the end of this section we have also provided code to return early after computing predictions from the probability vectors computed above. This will be useful at test time when we wish make predictions on each image without doing a full back propagation of the network which can be rather costly.

Step 1c: Back Propagation

First compute the error, δ_d, from the cross entropy cost function w.r.t. the parameters in the densely connected layer. You will then need to propagate this error through the subsampling and convolutional layer. Use MATLAB's kron function to upsample the error and propagate through the pooling layer.

Implementation tip: Using kron

You can upsample the error from an incoming layer to propagate through a mean-pooling layer quickly using MATLAB's kron function. This function takes the Kroneckor Tensor Product of two matrices. For example, suppose the pooling region was 2x2 on a 4x4 image. This means that the incoming error to the pooling layer will be of dimension 2x2 (assuming non-overlapping and contiguous pooling regions). The error must be upsampled from 2x2 to be 4x4. Since mean pooling is used, each error value contributes equally to the values in the region from which it came in the original 4x4 image. Let the incoming error to the pooling layer be given by

$$delta = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

If you use kron(delta, ones(2,2)), MATLAB will take the element by element product of each element in ones(2,2) with delta, as below:

$$\begin{pmatrix} 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 \\ 3 & 3 & 4 & 4 \\ 3 & 3 & 4 & 4 \end{pmatrix} \rightarrow \mathrm{kron}\left(\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}\right)$$

After the error has been upsampled, all that's left to be done to propagate through the pooling layer is to divide by the size of the pooling region. A basic implementation is shown below,

```
% Upsample the incoming error using kron
delta_pool = (1/poolDim^2) * kron(delta,ones(poolDim));
```

To propagate error through the convolutional layer, you simply need to multiply the incoming error by the derivative of the activation function as in the usual back propagation algorithm. Using these errors to compute the gradient w.r.t to each weight is a bit trickier since we have tied weights and thus many errors contribute to the gradient w.r.t. a single weight. We will discuss this in the next section.

Step 1d: Gradient Calculation

Compute the gradient for the densely connected weights and bias, W_d and b_d following the equations presented in multi-layer neural networks.

In order to compute the gradient with respect to each of the filters for a single training example (i.e. image) in the convolutional layer, you must first convolve the error term for that image-filter pair as computed in the previous step with the original training image. Again, use MATLAB's conv2 function with the 'valid' option to handle borders correctly. Make sure to flip the error matrix for that image-filter pair prior to the convolution as discussed in the simple convolution exercise. The final gradient for a given filter is the sum over the convolution of all images with the error for that image-filter pair.

The gradient w.r.t to the bias term for each filter in the convolutional layer is simply the sum of all error terms corresponding to the given filter.

Make sure to scale your gradients by the inverse size of the training set if you included this scale in the cost calculation otherwise your code will not pass the numerical gradient check.

Step 2: Gradient Check

Use the computeNumericalGradient function to check the cost and gradient of your convolutional network. We've provided a small sample set and toy network to run the numerical gradient check on.

Once your code passes the gradient check you're ready to move onto training a real network on the full dataset. Make sure to switch the DEBUG boolean to false in order not to run the gradient check again.

Step 3: Learn Parameters

Using a batch method such as L-BFGS to train a convolutional network of this size even on MNIST, a relatively small dataset, can be computationally slow. A single iteration of calculating the cost and gradient for the full training set can take several minutes or more. Thus you will use stochastic gradient descent (SGD) to learn the parameters of the network.

You will use SGD with momentum as described in Stochastic Gradient Descent. Implement the velocity vector and parameter vector update in minFuncSGD.m.

In this implementation of SGD we use a relatively heuristic method of annealing the learning rate for better convergence as learning slows down. We simply halve the learning rate after each epoch. As mentioned in Stochastic Gradient Descent, we also randomly shuffle the data before each epoch, which tends to provide better convergence.

Step 4: Test

With the convolutional network and SGD optimizer in hand, you are now ready to test the performance of the model. We've provided code at the end of cnnTrain.m to test the accuracy of your networks predictions on the MNIST test set.

Run the full function cnnTrain.m which will learn the parameters of you convolutional neural network over 3 epochs of the data. This shouldn't take more than 20 minutes. After 3 epochs, your networks accuracy on the MNIST test set should be above 96%.

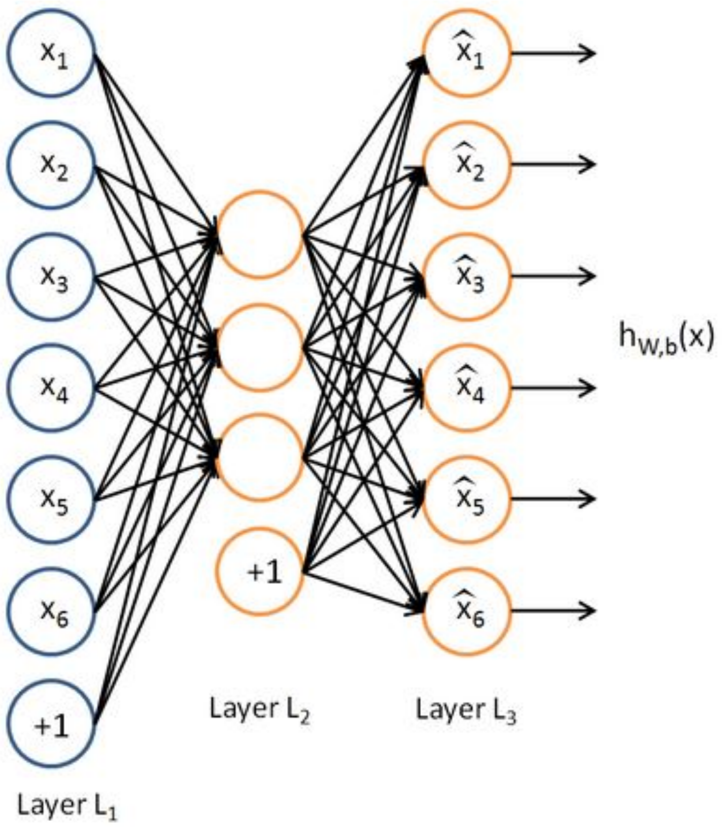
Congratulations, you've successfully implemented a Convolutional Neural Network!

Unsupervised Learning

Autoencoders

So far, we have described the application of neural networks to supervised learning, in which we have labeled training examples. Now suppose we have only a set of unlabeled training examples $\{x^{(1)}, x^{(2)}, x^{(3)}, \ldots\}$, where $x^{(i)} \in \Re^{n}$. An autoencoder neural network is an unsupervised learning algorithm that applies backpropagation, 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 $y \in \Re^{100}$. 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 $a^{(2)} \in \Re^{50}$, 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. In fact, this simple autoencoder often ends up learning a low-dimensional representation very similar to PCAs.

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 0. We would like to constrain the neurons to be inactive most of the time. This discussion assumes a sigmoid activation function. If you are using a tanh activation function, then we think of a neuron as being inactive when it outputs values close to -1.

Recall that $a_j^{(2)}$ denotes the activation of hidden unit j in the autoencoder. However, this notation doesn't make explicit what was the input x that led to that activation. Thus, we will write $a_j^{(2)}(x)$ to denote the activation of this hidden unit when the network is given a specific input x. Further, let

$$\hat\rho_j = \frac{1}{m}\sum_{i=1}^{m}\left[a_j^{(2)}(x^{(i)})\right]$$

be the average activation of hidden unit j (averaged over the training set). We would like to (approximately) enforce the constraint

$$\hat\rho_j = \rho,$$

where ρ is a sparsity parameter, typically a small value close to zero (say $\rho = 0.05$). In other words, we would like the average activation of each hidden neuron j to be close to 0.05 (say). To satisfy this constraint, the hidden unit's activations must mostly be near 0.

To achieve this, we will add an extra penalty term to our optimization objective that penalizes $\hat{\rho}_j$ deviating significantly from ρ. Many choices of the penalty term will give reasonable results. We will choose the following:

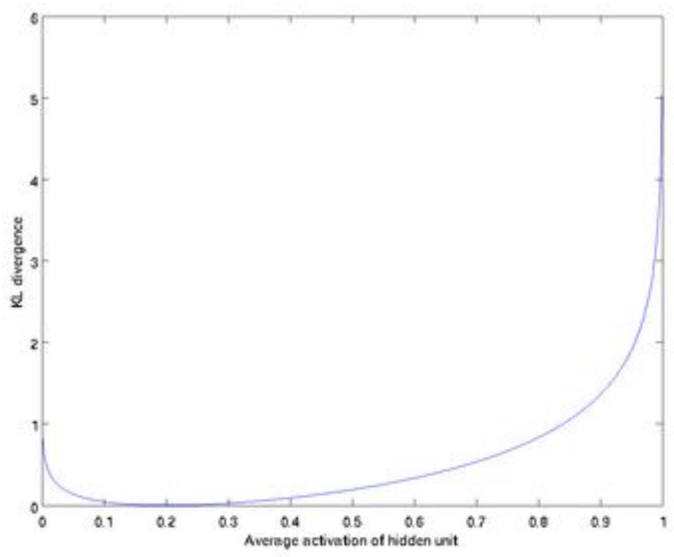
$$\sum_{j=1}^{s_2} \rho \log \frac{\rho}{\hat{\rho}_j} + (1-\rho) \log \frac{1-\rho}{1-\hat{\rho}_j}.$$

Here, s_2 is the number of neurons in the hidden layer, and the index j is summing over the hidden units in our network. If you are familiar with the concept of KL divergence, this penalty term is based on it, and can also be written

$$\sum_{j=1}^{s_2} \mathrm{KL}(\rho || \hat{\rho}_j),$$

where $\mathrm{KL}(\rho || \hat{\rho}_j) = \rho \log \frac{\rho}{\hat{\rho}_j} + (1-\rho) \log \frac{1-\rho}{1-\hat{\rho}_j}$ is the Kullback-Leibler (KL) divergence between a Bernoulli random variable with mean ρ and a Bernoulli random variable with mean $\hat{\rho}_j$. KL-divergence is a standard function for measuring how different two different distributions are. (If you've not seen KL-divergence before, don't worry about it; everything you need to know about it is contained in these notes.)

This penalty function has the property that $\mathrm{KL}(\rho || \hat{\rho}_j) = 0$ if $\hat{\rho}_j = \rho$, and otherwise it increases monotonically as $\hat{\rho}_j$ diverges from ρ. For example, in the figure below, we have set $\rho = 0.2$, and plotted $\mathrm{KL}(\rho || \hat{\rho}_j)$ for a range of values of $\hat{\rho}_j$:



We see that the KL-divergence reaches its minimum of 0 at $\hat\rho_j = \rho$, and blows up (it actually approaches ∞) as $\hat\rho_j$ approaches 0 or 1. Thus, minimizing this penalty term has the effect of causing $\hat\rho_j$ to be close to ρ.

Our overall cost function is now

$$J_{\rm sparse}(W,b) = J(W,b) + \beta \sum_{j=1}^{s_2} {\rm KL}(\rho || \hat\rho_j),$$

where $J(W,b)$ is as defined previously, and β controls the weight of the sparsity penalty term. The term $\hat\rho_j$ (implicitly) depends on W,balso, because it is the average activation of hidden unit j, and the activation of a hidden unit depends on the parameters W,b.

To incorporate the KL-divergence term into your derivative calculation, there is a simple-to-implement trick involving only a small change to your code. Specifically, where previously for the second layer ($l=2$), during backpropagation you would have computed

$$\delta^{(2)}_i = \left(\sum_{j=1}^{s_2} W^{(2)}_{ji} \delta^{(3)}_j \right) f'(z^{(2)}_i),$$

now instead compute

$$\delta_i^{(2)} = \left(\left(\sum_{j=1}^{s_2} W_{ji}^{(2)} \delta_j^{(3)} \right) + \beta \left(-\frac{\rho}{\hat\rho_i} + \frac{1-\rho}{1-\hat\rho_i} \right) \right) f'(z_i^{(2)}).$$

One subtlety is that you'll need to know $\hat\rho_i$ to compute this term. Thus, you'll need to compute a forward pass on all the training examples first to compute the average activations on the training set, before computing backpropagation on any example. If your training set is small enough to fit comfortably in computer memory (this will be the case for the programming assignment), you can compute forward passes on all your examples and keep the resulting activations in memory and compute the $\hat\rho_i$s. Then you can use your precomputed activations to perform backpropagation on all your examples. If your data is too large to fit in memory, you may have to scan through your examples computing a forward pass on each to accumulate (sum up) the activations and compute $\hat\rho_i$ (discarding the result of each forward pass after you have taken its activations $a_i^{(2)}$ into account for computing $\hat\rho_i$). Then after having computed $\hat\rho_i$, you'd have to redo the forward pass for each example so that you can do backpropagation on that example. In this latter case, you would end up computing a forward pass twice on each example in your training set, making it computationally less efficient.

The full derivation showing that the algorithm above results in gradient descent is beyond the scope of these notes. But if you implement the autoencoder using backpropagation modified this way, you will be performing gradient descent exactly on the objective $J_{\rm sparse}(W,b)$. Using the derivative checking method, you will be able to verify this for yourself as well.

Visualizing a Trained Autoencoder

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:

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

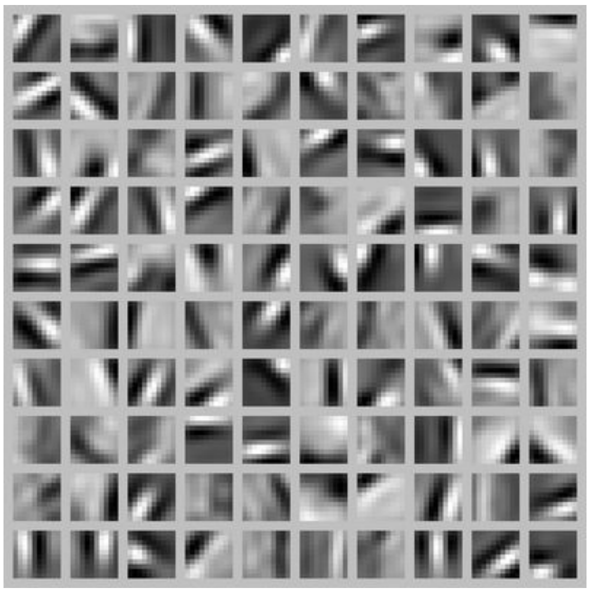
We will visualize the function computed by hidden unit i---which depends on the parameters $W_{ij}^{(1)}$ (ignoring the bias term for now)---using a 2D image. In particular, we think of $a_i^{(2)}$ as some non-linear feature of the input x. We ask: What input image x would cause $a_i^{(2)}$ to be maximally activated? (Less formally, what is the feature that hidden unit i is looking for?) 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 inputs1 we get the following result:



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.

1 The learned features 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.

PCA Whitening

Introduction

Principal Components Analysis (PCA) is a dimensionality reduction algorithm that can be used to significantly speed up your unsupervised feature learning algorithm. More importantly, understanding PCA will enable us to later implement whitening, which is an important pre-processing step for many algorithms.

Suppose you are training your algorithm on images. Then the input will be somewhat redundant, because the values of adjacent pixels in an image are highly correlated. Concretely, suppose we are training on 16x16 grayscale image patches. Then

$x \in \Re^{256}$ are 256 dimensional vectors, with one feature x_j corresponding to the intensity of each pixel. Because of the correlation between adjacent pixels, PCA will allow us to approximate the input with a much lower dimensional one, while incurring very little error.

Example and Mathematical Background

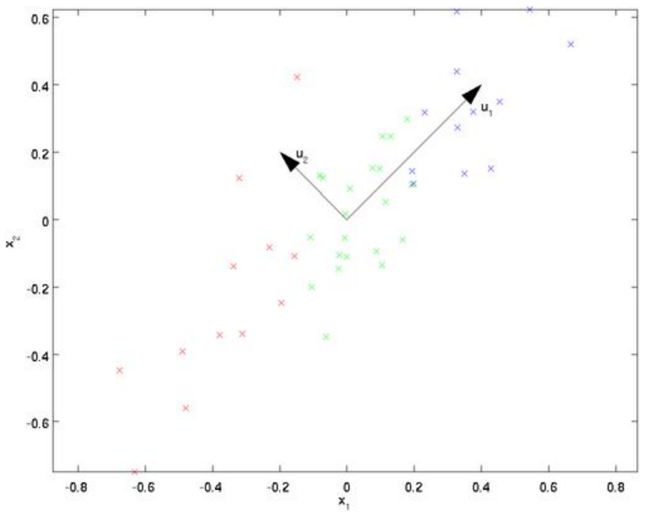
For our running example, we will use a dataset $\{x^{(1)}, x^{(2)}, \ldots, x^{(m)}\}$ with $n = 2$ dimensional inputs, so that $x^{(i)} \in \Re^2$. Suppose we want to reduce the data from 2 dimensions to 1. (In practice, we might want to reduce data from 256 to 50 dimensions, say; but using lower dimensional data in our example allows us to visualize the algorithms better.) Here is our dataset:

600px

This data has already been pre-processed so that each of the features x_1 and x_2 have about the same mean (zero) and variance.

For the purpose of illustration, we have also colored each of the points one of three colors, depending on their x_1 value; these colors are not used by the algorithm, and are for illustration only.

PCA will find a lower-dimensional subspace onto which to project our data. From visually examining the data, it appears that u_1 is the principal direction of variation of the data, and u_2 the secondary direction of variation:



I.e., the data varies much more in the direction u_1 than u_2. To more formally find the directions u_1 and u_2, we first compute the matrix Σ as follows:

$$\Sigma = \frac{1}{m} \sum_{i=1}^{m} (x^{(i)})(x^{(i)})^T.$$

If x has zero mean, then Σ is exactly the covariance matrix of x. (The symbol "Σ", pronounced "Sigma", is the standard notation for denoting the covariance matrix.

Unfortunately it looks just like the summation symbol, as in $\sum_{i=1}^{n} i$; but these are two different things.)

It can then be shown that u_1---the principal direction of variation of the data---is the top (principal) eigenvector of Σ, and u_2 is the second eigenvector.

Note: If you are interested in seeing a more formal mathematical derivation/justification of this result, see the CS229 (Machine Learning) lecture notes on PCA (link at bottom of this page). You won't need to do so to follow along this course, however.

You can use standard numerical linear algebra software to find these eigenvectors (see Implementation Notes). Concretely, let us compute the eigenvectors of Σ, and stack the eigenvectors in columns to form the matrix U:

$$U = \begin{bmatrix} | & | & & | \\ u_1 & u_2 & \cdots & u_n \\ | & | & & | \end{bmatrix}$$

Here, u_1 is the principal eigenvector (corresponding to the largest eigenvalue), u_2 is the second eigenvector, and so on. Also, let $\lambda_1, \lambda_2, \ldots, \lambda_n$ be the corresponding eigenvalues.

The vectors u_1 and u_2 in our example form a new basis in which we can represent the data. Concretely, let $x \in \Re^2$ be some training example. Then $u_1^T x$ is the length (magnitude) of the projection of x onto the vector u_1.

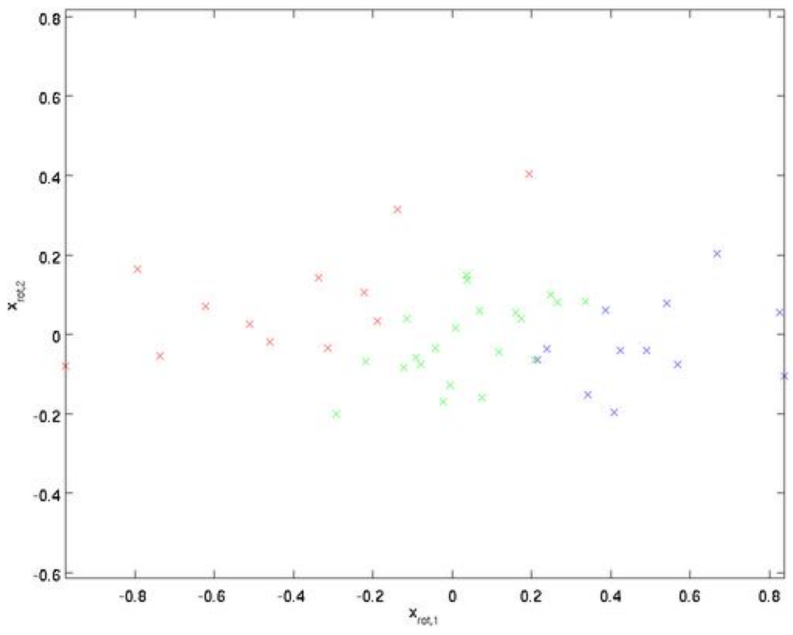
Similarly, $u_2^T x$ is the magnitude of x projected onto the vector u_2.

Rotating the Data

Thus, we can represent x in the (u_1, u_2)-basis by computing

$$x_{\rm rot} = U^T x = \begin{bmatrix} u_1^T x \\ u_2^T x \end{bmatrix}$$

(The subscript "rot" comes from the observation that this corresponds to a rotation (and possibly reflection) of the original data.) Lets take the entire training set, and compute $x_{\rm rot}^{(i)} = U^T x^{(i)}$ for every i. Plotting this transformed data $x_{\rm rot}$, we get:



This is the training set rotated into the u_1,u_2 basis. In the general case, $U^T x$ will be the training set rotated into the basis u_1,u_2, ...,u_n.

One of the properties of U is that it is an "orthogonal" matrix, which means that it satisfies $U^T U = U U^T = I$. So if you ever need to go from the rotated vectors $x_{\rm rot}$ back to the original data x, you can compute

$$x = U x_{\rm rot},$$

because $U x_{\rm rot} = U U^T x = x$.

Reducing the Data Dimension

We see that the principal direction of variation of the data is the first dimension $x_{\rm rot,1}$ of this rotated data. Thus, if we want to reduce this data to one dimension, we can set

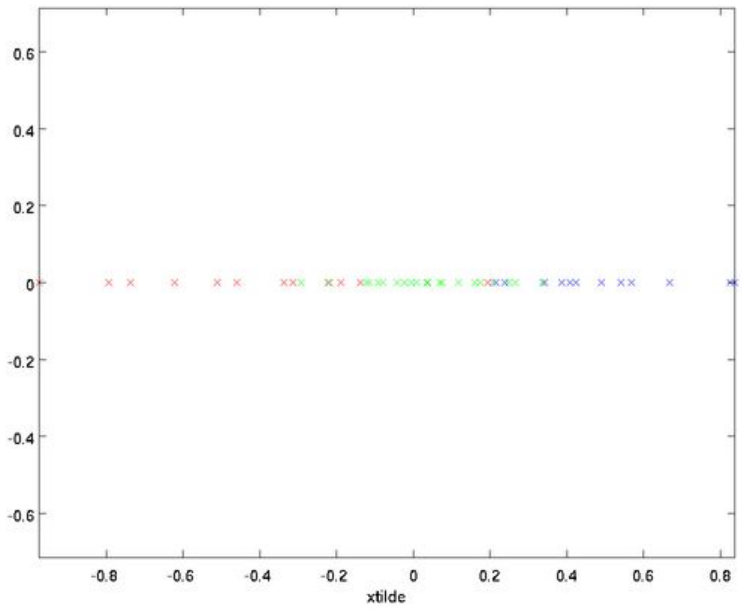
$$\tilde{x}^{(i)} = x_{\rm rot,1}^{(i)} = u_1^T x^{(i)} \in \Re.$$

More generally, if $x \in \Re^n$ and we want to reduce it to a k dimensional representation $\tilde{x} \in \Re^k$ (where $k < n$), we would take the first $k$components of $x_{\rm rot}$, which correspond to the top k directions of variation.

Another way of explaining PCA is that $x_{\rm rot}$ is an n dimensional vector, where the first few components are likely to be large (e.g., in our example, we saw that $x_{\rm rot,1}^{(i)} = u_1^T x^{(i)}$ takes reasonably large values for most examples i), and the later components are likely to be small (e.g., in our example, $x_{\rm rot,2}^{(i)} = u_2^T x^{(i)}$ was more likely to be small). What PCA does it it drops the the later (smaller) components of $x_{\rm rot}$, and just approximates them with 0's. Concretely, our definition of $\tilde{x}$ can also be arrived at by using an approximation to $x_{\rm rot}$ where all but the first components are zeros. In other words, we have:

$$\tilde{x} = \begin{bmatrix} x_{\rm rot,1} \\ \vdots \\ x_{\rm rot,k} \\ 0 \\ \vdots \\ 0 \end{bmatrix} \approx \begin{bmatrix} x_{\rm rot,1} \\ \vdots \\ x_{\rm rot,k} \\ x_{\rm rot,k+1} \\ \vdots \\ x_{\rm rot,n} \end{bmatrix} = x_{\rm rot}$$

In our example, this gives us the following plot of $\tilde{x}$ (using $n = 2, k = 1$):



However, since the final $n-k$ components of $\tilde{x}$ as defined above would always be zero, there is no need to keep these zeros around, and so we define $\tilde{x}$ as a k-dimensional vector with just the first k (non-zero) components.

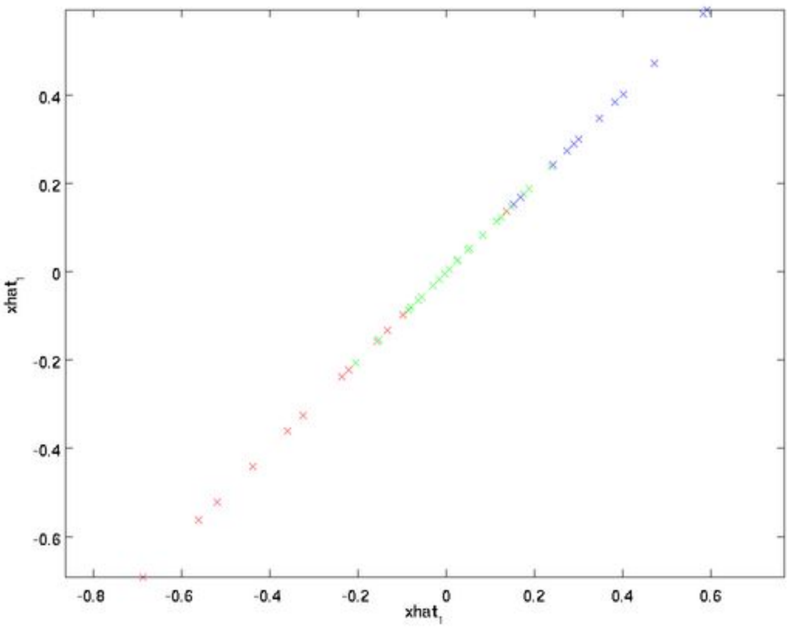
This also explains why we wanted to express our data in the $u_1, u_2, \ldots, u_n$ basis: Deciding which components to keep becomes just keeping the top k components. When we do this, we also say that we are "retaining the top k PCA (or principal) components."

Recovering an Approximation of the Data

Now, $\tilde{x} \in \Re^k$ is a lower-dimensional, "compressed" representation of the original $x \in \Re^n$. Given $\tilde{x}$, how can we recover an approximation $\hat{x}$to the original value of x? From an earlier section, we know that $x = U x_{\rm rot}$. Further, we can think of $\tilde{x}$ as an approximation to $x_{\rm rot}$, where we have set the last $n-k$ components to zeros. Thus, given $\tilde{x} \in \Re^k$, we can pad it out with $n-k$ zeros to get our approximation to$x_{\rm rot} \in \Re^n$. Finally, we pre-multiply by U to get our approximation to x. Concretely, we get

$$\hat{x} = U \begin{bmatrix} \tilde{x}_1 \\ \vdots \\ \tilde{x}_k \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \sum_{i=1}^{k} u_i \tilde{x}_i.$$

The final equality above comes from the definition of U given earlier. (In a practical implementation, we wouldn't actually zero pad $\tilde{x}$and then multiply by U, since that would mean multiplying a lot of things by zeros; instead, we'd just multiply $\tilde{x} \in \Re^k$ with the first $k$columns of U as in the final expression above.) Applying this to our dataset, we get the following plot for $\hat{x}$:



We are thus using a 1 dimensional approximation to the original dataset.

If you are training an autoencoder or other unsupervised feature learning algorithm, the running time of your algorithm will depend on the dimension of the input. If you feed $\tilde{x} \in \Re^k$ into your learning algorithm instead of x, then you'll be training on a lower-dimensional input, and thus your algorithm might run significantly faster. For many datasets, the lower dimensional $\tilde{x}$ representation can be an extremely good approximation to the original, and using PCA this way can significantly speed up your algorithm while introducing very little approximation error.

Number of components to retain

How do we set k; i.e., how many PCA components should we retain? In our simple 2 dimensional example, it seemed natural to retain 1 out of the 2 components, but for higher dimensional data, this decision is less trivial. If k is too large, then we won't be compressing the data much; in the limit of $k = n$, then we're just using the original data (but rotated into a different basis). Conversely, if k is too small, then we might be using a very bad approximation to the data.

To decide how to set k, we will usually look at the percentage of variance retained for different values of k. Concretely, if $k = n$, then we have an exact approximation to the data, and we say that 100% of the variance is retained. I.e., all of the variation of the original data is retained. Conversely, if $k = 0$, then we are approximating all the data with the zero vector, and thus 0% of the variance is retained.

More generally, let $\lambda_1, \lambda_2, \ldots, \lambda_n$ be the eigenvalues of Σ (sorted in decreasing order), so that λ_j is the eigenvalue corresponding to the eigenvector u_j. Then if we retain k principal components, the percentage of variance retained is given by:

$$\frac{\sum_{j=1}^{k} \lambda_j}{\sum_{j=1}^{n} \lambda_j}.$$

In our simple 2D example above, $\lambda_1 = 7.29$, and $\lambda_2 = 0.69$. Thus, by keeping only $k = 1$ principal components, we retained $7.29/(7.29 + 0.69) = 0.913$, or 91.3% of the variance.

A more formal definition of percentage of variance retained is beyond the scope of these notes. However, it is possible to show that $\lambda_j = \sum_{i=1}^{m} x_{\text{rot},j}^2$. Thus, if $\lambda_j \approx 0$, that shows that $x_{\text{rot},j}$ is usually near 0 anyway, and we lose relatively little by approximating it with a constant 0. This also explains why we retain the top principal components (corresponding to the larger values of λ_j) instead of the bottom ones. The top principal components $x_{\text{rot},j}$ are the ones that're more variable and that take on larger values, and for which we would incur a greater approximation error if we were to set them to zero.

In the case of images, one common heuristic is to choose k so as to retain 99% of the variance. In other words, we pick the smallest value of k that satisfies

$$\frac{\sum_{j=1}^{k}\lambda_j}{\sum_{j=1}^{n}\lambda_j} \geq 0.99.$$

Depending on the application, if you are willing to incur some additional error, values in the 90-98% range are also sometimes used. When you describe to others how you applied PCA, saying that you chose k to retain 95% of the variance will also be a much more easily interpretable description than saying that you retained 120 (or whatever other number of) components.

PCA on Images

For PCA to work, usually we want each of the features $x_1, x_2, \ldots, x_n$ to have a similar range of values to the others (and to have a mean close to zero). If you've used PCA on other applications before, you may therefore have separately pre-processed each feature to have zero mean and unit variance, by separately estimating the mean and variance of each feature x_j. However, this isn't the pre-processing that we will apply to most types of images. Specifically, suppose we are training our algorithm on natural images, so that x_j is the value of pixel j. By "natural images," we informally mean the type of image that a typical animal or person might see over their lifetime.

Note: Usually we use images of outdoor scenes with grass, trees, etc., and cut out small (say 16x16) image patches randomly from these to train the algorithm. But in practice most feature learning algorithms are extremely robust to the exact type of image it is trained on, so most images taken with a normal camera, so long as they aren't excessively blurry or have strange artifacts, should work.

When training on natural images, it makes little sense to estimate a separate mean and variance for each pixel, because the statistics in one part of the image should (theoretically) be the same as any other. This property of images is called stationarity.

In detail, in order for PCA to work well, informally we require that (i) The features have approximately zero mean, and (ii) The different features have similar variances to each other. With natural images, (ii) is already satisfied even without variance normalization, and so we won't perform any variance normalization. (If you are training on audio data---say, on spectrograms---or on text data---say, bag-of-word vectors---we will usually not perform variance normalization either.) In fact, PCA is invariant to the scaling of the data, and will return the same eigenvectors regardless of the scaling of the input. More formally, if you multiply each feature vector x by some positive number (thus scaling every feature in every training example by the same number), PCA's output eigenvectors will not change.

So, we won't use variance normalization. The only normalization we need to perform then is mean normalization, to ensure that the features have a mean around zero. Depending on the application, very often we are not interested in how bright the overall input image is. For example, in object recognition tasks, the overall brightness of the image doesn't affect what objects there are in the image. More formally, we are not interested in the mean intensity value of an image patch; thus, we can subtract out this value, as a form of mean normalization.

Concretely, if $x^{(i)} \in \Re^n$ are the (grayscale) intensity values of a 16x16 image patch ($n = 256$), we might normalize the intensity of each image $x^{(i)}$ as follows:

$$\mu^{(i)} := \frac{1}{n} \sum_{j=1}^{n} x_j^{(i)}$$

$x_j^{(i)} := x_j^{(i)} - \mu^{(i)}$, for all j

Note that the two steps above are done separately for each image $x^{(i)}$, and that $\mu^{(i)}$ here is the mean intensity of the image $x^{(i)}$. In particular, this is not the same thing as estimating a mean value separately for each pixel x_j.

If you are training your algorithm on images other than natural images (for example, images of handwritten characters, or images of single isolated objects centered against a white background), other types of normalization might be worth considering, and the best choice may be application dependent. But when training on natural images, using the per-image mean normalization method as given in the equations above would be a reasonable default.

References

http://cs229.stanford.edu

Whitening

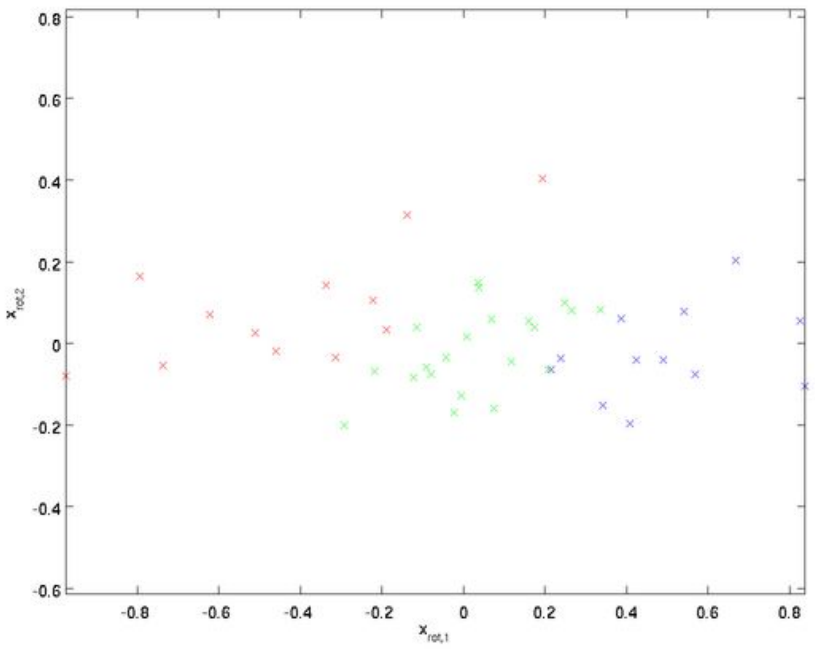
Introduction

We have used PCA to reduce the dimension of the data. There is a closely related preprocessing step called whitening (or, in some other literatures, sphering) which is needed for some algorithms. If we are training on images, the raw input is redundant, since adjacent pixel values are highly correlated. The goal of whitening is to make the input less redundant; more formally, our desiderata are that our learning algorithms sees a training input where (i) the features are less correlated with each other, and (ii) the features all have the same variance.

2D example

We will first describe whitening using our previous 2D example. We will then describe how this can be combined with smoothing, and finally how to combine this with PCA.

How can we make our input features uncorrelated with each other? We had already done this when computing $x_{\rm rot}^{(i)} = U^T x^{(i)}$. Repeating our previous figure, our plot for $x_{\rm rot}$ was:



The covariance matrix of this data is given by:

$$\begin{bmatrix} 7.29 & 0 \\ 0 & 0.69 \end{bmatrix}.$$

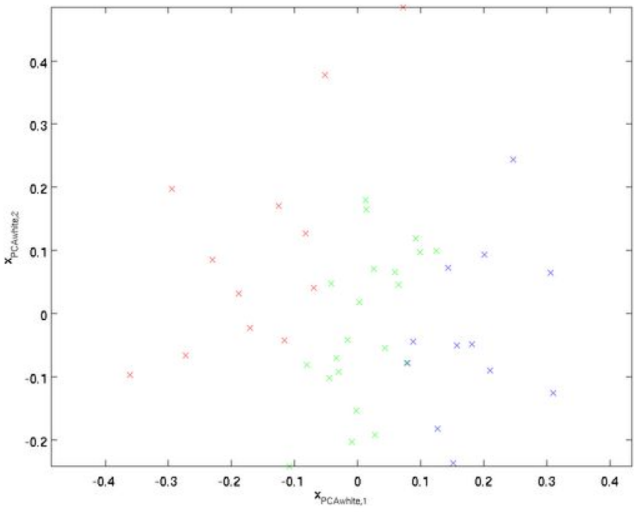
(Note: Technically, many of the statements in this section about the "covariance" will be true only if the data has zero mean. In the rest of this section, we will take this assumption as implicit in our statements. However, even if the data's mean isn't exactly zero, the intuitions we're presenting here still hold true, and so this isn't something that you should worry about.)

It is no accident that the diagonal values are λ_1 and λ_2. Further, the off-diagonal entries are zero; thus, $x_{\rm rot,1}$ and $x_{\rm rot,2}$ are uncorrelated, satisfying one of our desiderata for whitened data (that the features be less correlated).

To make each of our input features have unit variance, we can simply rescale each feature $x_{\rm rot,\it i}$ by $1/\sqrt{\lambda_i}$. Concretely, we define our whitened data $x_{\rm PCAwhite} \in \Re^n$ as follows:

$$x_{\rm PCAwhite,\it i} = \frac{x_{\rm rot,\it i}}{\sqrt{\lambda_i}}.$$

Plotting $x_{\rm PCAwhite}$, we get:



This data now has covariance equal to the identity matrix I. We say that $x_{\rm PCAwhite}$ is our PCA whitened version of the data: The different components of $x_{\rm PCAwhite}$ are uncorrelated and have unit variance.

Whitening combined with dimensionality reduction. If you want to have data that is whitened and which is lower dimensional than the original input, you can also optionally keep only the top k components of $x_{\rm PCAwhite}$. When we combine PCA whitening with regularization (described later), the last few components of

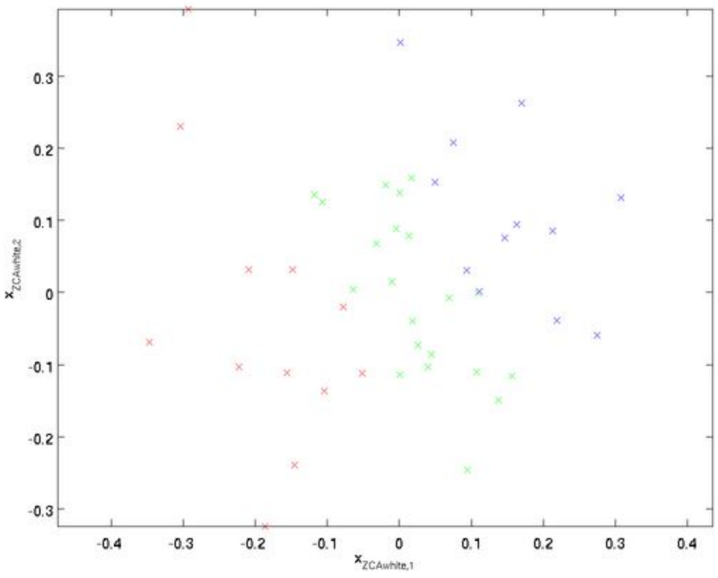
$x_{\rm PCAwhite}$ will be nearly zero anyway, and thus can safely be dropped.

ZCA Whitening

Finally, it turns out that this way of getting the data to have covariance identity I isn't unique. Concretely, if R is any orthogonal matrix, so that it satisfies $RR^T = R^TR = I$ (less formally, if R is a rotation/reflection matrix), then $R\,x_{\rm PCAwhite}$ will also have identity covariance. In ZCA whitening, we choose $R = U$. We define

$x_{\rm ZCAwhite} = U x_{\rm PCAwhite}$

Plotting $x_{\rm ZCAwhite}$, we get:



It can be shown that out of all possible choices for R, this choice of rotation causes $x_{\rm ZCAwhite}$ to be as close as possible to the original input data x.

When using ZCA whitening (unlike PCA whitening), we usually keep all n dimensions of the data, and do not try to reduce its dimension.

Regularizaton

When implementing PCA whitening or ZCA whitening in practice, sometimes some of the eigenvalues λ_i will be numerically close to 0, and thus the scaling step where we divide by $\sqrt{\lambda_i}$ would involve dividing by a value close to zero; this may cause the data to blow up (take on large values) or otherwise be numerically unstable. In practice, we therefore implement this scaling step using a small amount of regularization, and add a small constant ϵ to the eigenvalues before taking their square root and inverse:

$$x_{\rm PCAwhite,\it i} = \frac{x_{{\rm rot},i}}{\sqrt{\lambda_i + \epsilon}}.$$

When x takes values around $[-1, 1]$, a value of $\epsilon \approx 10^{-5}$ might be typical.

For the case of images, adding ϵ here also has the effect of slightly smoothing (or low-pass filtering) the input image. This also has a desirable effect of removing aliasing artifacts caused by the way pixels are laid out in an image, and can improve the features learned (details are beyond the scope of these notes).

ZCA whitening is a form of pre-processing of the data that maps it from x to $x_{\rm ZCAwhite}$. It turns out that this is also a rough model of how the biological eye (the retina) processes images. Specifically, as your eye perceives images, most adjacent "pixels" in your eye will perceive very similar values, since adjacent parts of an image tend to be highly correlated in intensity. It is thus wasteful for your eye to have to transmit every pixel separately (via your optic nerve) to your brain. Instead, your retina performs a decorrelation operation (this is done via retinal neurons that compute a function called "on center, off surround/off center, on surround") which is similar to that performed by ZCA. This results in a less redundant representation of the input image, which is then transmitted to your brain.

Implementing PCA Whitening

In this section, we summarize the PCA, PCA whitening and ZCA whitening algorithms, and also describe how you can implement them using efficient linear algebra libraries.

First, we need to ensure that the data has (approximately) zero-mean. For natural images, we achieve this (approximately) by subtracting the mean value of each image patch.

We achieve this by computing the mean for each patch and subtracting it for each patch. In Matlab, we can do this by using

```
avg = mean(x, 1);         % Compute the mean pixel intensity value separately for each patch.

x = x - repmat(avg, size(x, 1), 1);
```

Next, we need to compute $\Sigma = \frac{1}{m} \sum_{i=1}^{m} (x^{(i)})(x^{(i)})^T$. If you're implementing this in Matlab (or even if you're implementing this in C++, Java, etc., but have access to an efficient linear algebra library), doing it as an explicit sum is inefficient. Instead, we can compute this in one fell swoop as

```
sigma = x * x' / size(x, 2);
```

(Check the math yourself for correctness.) Here, we assume that x is a data structure that contains one training example per column (so,x is a n-by-m matrix).

Next, PCA computes the eigenvectors of Σ. One could do this using the Matlab eig function. However, because Σ is a symmetric positive semi-definite matrix, it is more numerically reliable to do this using the svd function. Concretely, if you implement

```
[U,S,V] = svd(sigma);
```

then the matrix U will contain the eigenvectors of Sigma (one eigenvector per column, sorted in order from top to bottom eigenvector), and the diagonal entries of the matrix S will contain the corresponding eigenvalues (also sorted in decreasing order). The matrix V will be equal to transpose of U, and can be safely ignored.

(Note: The svd function actually computes the singular vectors and singular values of a matrix, which for the special case of a symmetric positive semi-definite matrix---which is all that we're concerned with here---is equal to its eigenvectors and eigenvalues. A full discussion of singular vectors vs. eigenvectors is beyond the scope of these notes.)

Finally, you can compute $x_{\rm rot}$ and $\tilde{x}$ as follows:

```
xRot = U' * x;              % rotated version of the data.

xTilde = U(:,1:k)' * x; % reduced dimension representation of the data,

                              % where k is the number of eigenvectors to keep
```

This gives your PCA representation of the data in terms of $\tilde{x} \in \Re^k$. Incidentally, if x is a n -by- m matrix containing all your training data, this is a vectorized implementation, and the expressions above work too for computing xrot and $\tilde{x}$ for your entire training set all in one go. The resulting xrot and $\tilde{x}$ will have one column corresponding to each training example.

To compute the PCA whitened data $x_{\rm PCAwhite}$, use

```
xPCAwhite = diag(1./sqrt(diag(S) + epsilon)) * U' * x;
```

Since S's diagonal contains the eigenvalues λ_i, this turns out to be a compact way of computing $x_{{\rm PCAwhite},i} = \frac{x_{{\rm rot},i}}{\sqrt{\lambda_i}}$ simultaneously for all i.

Finally, you can also compute the ZCA whitened data $x_{\rm ZCAwhite}$ as:

```
xZCAwhite = U * diag(1./sqrt(diag(S) + epsilon)) * U' * x;
```

Exercise: PCA Whitening

PCA and Whitening on natural images

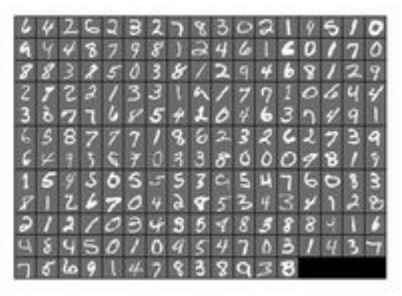
In this exercise, you will implement PCA, PCA whitening and ZCA whitening, and apply them to image patches taken from natural images.

You will build on the MATLAB starter code which we have provided in pca_exercise.zip. You need only write code at the places indicated by "YOUR CODE HERE" in the files. The only file you need to modify is pca_gen.m.

Step 0: Prepare data

Step 0a: Load data

The starter code contains code to load a set of MNIST images. The raw patches will look something like this:



These patches are stored as column vectors $x^{(i)} \in \mathbb{R}^{144}$ in the 144×10000 matrix x.

Step 0b: Zero mean the data

First, for each image patch, compute the mean pixel value and subtract it from that image, this centering the image around zero. You should compute a different mean value for each image patch.

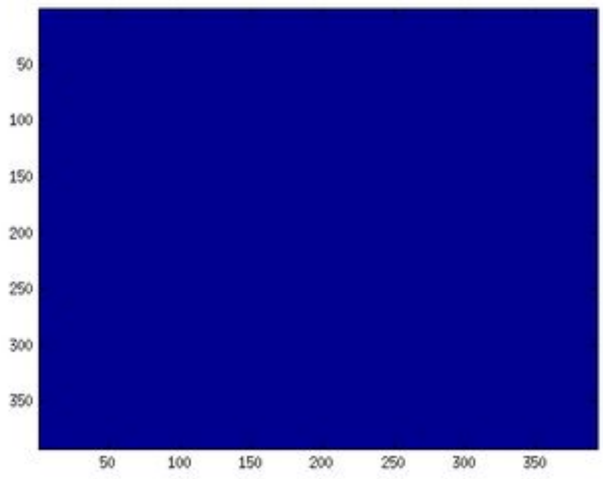
Step 1: Implement PCA

Step 1a: Implement PCA

In this step, you will implement PCA to obtain xrot, the matrix in which the data is "rotated" to the basis comprising the principal components (i.e. the eigenvectors of Σ). Note that in this part of the exercise, you should not whiten the data.

Step 1b: Check covariance

To verify that your implementation of PCA is correct, you should check the covariance matrix for the rotated data xrot. PCA guarantees that the covariance matrix for the rotated data is a diagonal matrix (a matrix with non-zero entries only along the main diagonal). Implement code to compute the covariance matrix and verify this property. One way to do this is to compute the covariance matrix, and visualise it using the MATLAB command imagesc. The image should show a coloured diagonal line against a blue background. For this dataset, because of the range of the diagonal entries, the diagonal line may not be apparent, so you might get a figure like the one show below, but this trick of visualizing using imagesc will come in handy later in this exercise.



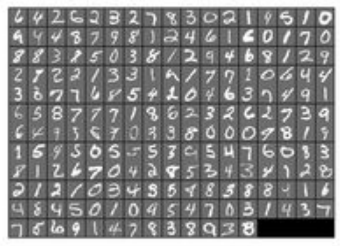
Step 2: Find number of components to retain

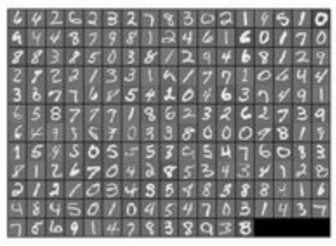
Next, choose k, the number of principal components to retain. Pick k to be as small as possible, but so that at least 99% of the variance is retained. In the step after this, you will discard all but the top k principal components, reducing the dimension of the original data to k.

Step 3: PCA with dimension reduction

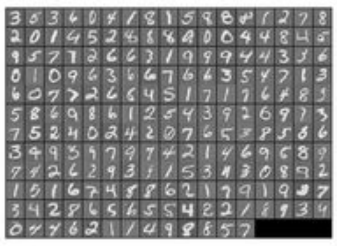
Now that you have found k, compute $\tilde{x}$, the reduced-dimension representation of the data. This gives you a representation of each image patch as a k dimensional vector instead of a 144 dimensional vector. If you are training a sparse autoencoder or other algorithm on this reduced-dimensional data, it will run faster than if you were training on the original 144 dimensional data.

To see the effect of dimension reduction, go back from $\tilde{x}$ to produce the matrix $\hat{x}$, the dimension-reduced data but expressed in the original 144 dimensional space of image patches. Visualise $\hat{x}$ and compare it to the raw data, x. You will observe that there is little loss due to throwing away the principal components that correspond to dimensions with low variation. For comparison, you may also wish to generate and visualise $\hat{x}$ for when only 90% of the variance is retained.


Raw images


PCA dimension-reduced images

(99% variance)


PCA dimension-reduced images

(90% variance)

Step 4: PCA with whitening and regularization

Step 4a: Implement PCA with whitening and regularization

Now implement PCA with whitening and regularization to produce the matrix xPCAWhite. Use the following parameter value:

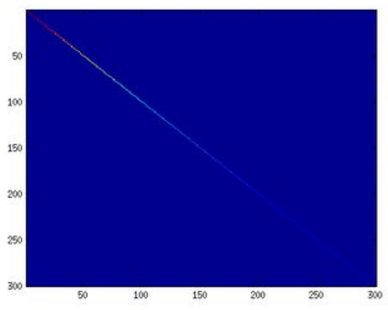
```
epsilon = 0.1
```

Step 4b: Check covariance

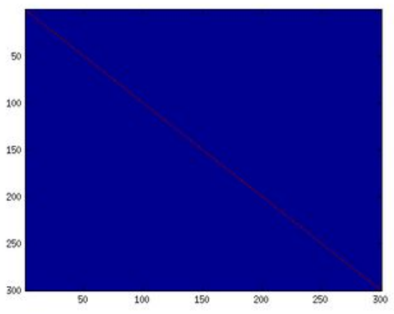
Similar to using PCA alone, PCA with whitening also results in processed data that has a diagonal covariance matrix. However, unlike PCA alone, whitening additionally ensures that the diagonal entries are equal to 1, i.e. that the covariance matrix is the identity matrix.

That would be the case if you were doing whitening alone with no regularization. However, in this case you are whitening with regularization, to avoid numerical/etc. problems associated with small eigenvalues. As a result of this, some of the diagonal entries of the covariance of your xPCAwhite will be smaller than 1.

To verify that your implementation of PCA whitening with and without regularization is correct, you can check these properties. Implement code to compute the covariance matrix and verify this property. (To check the result of PCA without whitening, simply set epsilon to 0, or close to 0, say 1e-10). As earlier, you can visualise the covariance matrix with imagesc. When visualised as an image, for PCA whitening without regularization you should see a red line across the diagonal (corresponding to the one entries) against a blue background (corresponding to the zero entries); for PCA whitening with regularization you should see a red line that slowly turns blue across the diagonal (corresponding to the 1 entries slowly becoming smaller).



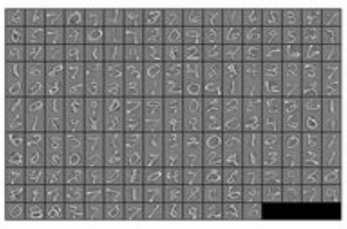
Covariance for PCA whitening with regularization



Covariance for PCA whitening without regularization

Step 5: ZCA whitening

Now implement ZCA whitening to produce the matrix xZCAWhite. Visualize xZCAWhite and compare it to the raw data, x. You should observe that whitening results in, among other things, enhanced edges. Try repeating this with epsilon set to 1, 0.1, and 0.01, and see what you obtain. The example shown below (left image) was obtained with epsilon = 0.1.



ZCA whitened images

Raw images

ICA

Introduction

If you recall, in sparse coding, we wanted to learn an over-complete basis for the data. In particular, this implies that the basis vectors that we learn in sparse coding will not be linearly independent. While this may be desirable in certain situations, sometimes we want to learn a linearly independent basis for the data. In independent component analysis (ICA), this is exactly what we want to do. Further, in ICA, we want to learn not just any linearly independent basis, but an orthonormal basis for the data. (An orthonormal basis is a basis $(\phi_1, \ldots \phi_n)$ such that $\phi_i \cdot \phi_j = 0$ if $i \neq j$ and 1 if i = j).

Like sparse coding, independent component analysis has a simple mathematical formulation. Given some data x, we would like to learn a set of basis vectors which we represent in the columns of a matrix W, such that, firstly, as in sparse coding, our features are sparse; and secondly, our basis is an orthonormal basis. (Note that while in sparse coding, our matrix A was for mapping features s to raw data, in independent component analysis, our matrix W works in the opposite direction, mapping raw data x to features instead). This gives us the following objective function:

$$J(W) = \|Wx\|_1$$

This objective function is equivalent to the sparsity penalty on the features s in sparse coding, since Wx is precisely the features that represent the data. Adding in the orthonormality constraint gives us the full optimization problem for independent component analysis:

$$\begin{array}{rl} \text{minimize} & \|Wx\|_1 \\ \text{s.t.} & WW^T = I \end{array}$$

As is usually the case in deep learning, this problem has no simple analytic solution, and to make matters worse, the orthonormality constraint makes it slightly more difficult to optimize for the objective using gradient descent - every iteration of gradient descent must be followed by a step that maps the new basis back to the space of orthonormal bases (hence enforcing the constraint).

In practice, optimizing for the objective function while enforcing the orthonormality constraint (as described in Orthonormal ICAsection below) is feasible but slow. Hence, the use of orthonormal ICA is limited to situations where it is important to obtain an orthonormal basis (TODO: what situations) .

Orthonormal ICA

The orthonormal ICA objective is:

$$\begin{array}{rl} \text{minimize} & \|Wx\|_1 \\ \text{s.t.} & WW^T = I \end{array}$$

Observe that the constraint WWT = I implies two other constraints.

Firstly, since we are learning an orthonormal basis, the number of basis vectors we learn must be less than the dimension of the input. In particular, this means that we cannot learn over-complete bases as we usually do in sparse coding.

Secondly, the data must be ZCA whitened with no regularization (that is, with ϵ set to 0). (TODO Why must this be so?)

Hence, before we even begin to optimize for the orthonormal ICA objective, we must ensure that our data has been whitened, and that we are learning an under-complete basis.

Following that, to optimize for the objective, we can use gradient descent, interspersing gradient descent steps with projection steps to enforce the orthonormality constraint. Hence, the procedure will be as follows:

Repeat until done:

$$W \leftarrow W - \alpha \nabla_W \|Wx\|_1$$

$W \leftarrow \text{proj}_U W$ where U is the space of matrices satisfying WWT = I

In practice, the learning rate α is varied using a line-search algorithm to speed up the descent, and the projection step is achieved by setting $W \leftarrow (WW^T)^{-\frac{1}{2}} W$, which can actually be seen as ZCA whitening (TODO explain how it is like ZCA whitening).

Topographic ICA

Just like sparse coding, independent component analysis can be modified to give a topographic variant by adding a topographic cost term.

RICA

ICA Summary

Independent Component Analysis (ICA) allows us to generate sparse representations of whitened data by the following formula:

$$\begin{aligned} \text{minimize} \quad & \|Wx\|_1 \\ \text{s.t.} \quad & WW^T = I \end{aligned}$$

where W is our weight matrix and x is our input. In ICA, we minimize the L1 penalty (sparsity) of our hidden representation, Wx, while maintaining an orthonormal constraint for our weight matrix. The orthonormal constraint exists to ensure that our uncorrelated data remains uncorrelated within our feature representation. In other words, an orthonormal transformation of whitened data remains white.

The orthonormal constraint in ICA presents some drawbacks to the algorithm. Namely, difficulties arise when the number of features (rows of W matrix), exceed the dimensionality of input, x. Optimization also becomes more difficult with hard constraints, and thus requires longer training. How could we speed this up? What if the dimensionality of our data is too large to be whitened? Keep in mind, if $x \in R^n$ it requires an $n \times n$ whitening matrix.

RICA

One algorithm called Reconstruction ICA (RICA), was designed to overcome the drawbacks of ICA by replacing ICA's orthonormality constraint with a soft reconstruction penalty.

$$\min_W \quad \lambda \|Wx\|_1 + \frac{1}{2} \left\| W^T W x - x \right\|_2^2$$

To help understand the motivation behind this, we see that we can get a perfect reconstruction when the features are not over-complete. To achieve this, we constrain WTW = I. It is also possible to recover ICA from RICA when features are not over-complete, data is whitened, and λ goes to infinity; at this point, perfect reconstruction becomes a hard constraint. Now that we have a reconstructive penalty in our objective and no hard constraints, we are able to scale up to over-complete features. However, will the result still be reasonable when we are using an over-complete basis? To answer this, we move to another common model, the sparse autoencoder.

To better interpret what happens when we move to an over-complete case, let's revisit sparse autoencoders. The objective is listed below:

$$\min_{W} \quad \lambda \left\|\sigma\left(Wx\right)\right\|_1 + \frac{1}{2}\left\|\sigma\left(W^T\sigma\left(Wx\right)\right) - x\right\|_2^2$$

There are different variations of autoencoders, but for the sake of consistency, this formula uses an L1 sparsity penalty and has a tied reconstruction matrix W. The only difference between this sparse autoencoder and RICA is the sigmoid non-linearity. Now, looking at the reconstructive penalty from the auto-encoder perspective, we can see that the reconstructive penalty acts as a degeneracy control; that is, the reconstructive penalty allows for the sparsest possible representation by ensuring that the filter matrix does not learn copies or redundant features. Thus we can see that RICA in the over-complete case is the same as a sparse autoencoder with an L1 sparsity constraint and without non-linearity. This allows RICA to scale to over-complete basis and be optimized with backprop like sparse auto-encoders. RICA has also been shown to be more robust to non-whitened data, which is again more similar to auto-encoder behavior.

Exercise: RICA

In this exercise, you will implement a one-layer RICA network and apply them to MNIST images.

You will build on MATLAB starter code which we have provided in https://github.com/amaas/stanford_dl_ex. You need only write code at places indicated by "YOUR CODE HERE". You will modify the files softICACost.m and zca2.m

Step 0: Prerequisites

Step 0a: Read runSoftICA.m

The file runSoftICA.m is the "main" script. It handles loading data, preprocessing it, and calling minFunc with the appropriate parameters. Be sure to understand how this file works before moving further.

Step 0b: Implement zca2.m

Implement the ZCA transform in zca2.m. You should be able to copy and paste your code from Exercise: PCA Whitening if you have successfully completed that exercise.

Step 1: RICA cost and gradient

First, let us derive the gradient of the RICA reconstruction cost using the backpropagation idea.

Step 1a: Deriving gradient using Backpropagation

Recall the RICA reconstruction cost term: $\|W^TWx - x\|_2^2$ where W is the weight matrix and x is the input.

We would like to find $\nabla_W\|W^TWx - x\|_2^2$ - the derivative of the term with respect to the weight matrix, rather than the input as in the earlier two examples. We will still proceed similarly though, seeing this term as an instantiation of a neural network:

400px

The weights and activation functions of this network are as follows:

Layer	Weight	Activation functionf
1	W	f(zi) = zi
2	WT	f(zi) = zi
3	I	f(zi) = zi − xi
4	N/A	$f(z_i) = z_i^2$

To have J(z(4)) = F(x), we can set $J(z^{(4)}) = \sum_k J(z_k^{(4)})$.

Now that we can see F as a neural network, we can try to compute the gradient $\nabla_W F$. However, we now face the difficulty that W appears twice in the network. Fortunately, it turns out that if W appears multiple times in the network, the gradient with respect to W is simply the sum of gradients for each instance of W in the network (you may wish to work out a formal proof of this fact to convince yourself). With this in mind, we will proceed to work out the deltas first:

Layer	Derivative of activation functionf'	Delta	Input z to this layer
4	f'(zi) = 2zi	f'(zi) = 2zi	(WTWx − x)

3	f'(zi) = 1	$\left(I^T\delta^{(4)}\right)\bullet 1$	WTWx
2	f'(zi) = 1	$\left((W^T)^T\delta^{(3)}\right)\bullet 1$	Wx
1	f'(zi) = 1	$\left(W^T\delta^{(2)}\right)\bullet 1$	x

To find the gradients with respect to W, first we find the gradients with respect to each instance of W in the network.

With respect to WT:

$$\begin{aligned}\nabla_{W^T}F &= \delta^{(3)}a^{(2)T} \\ &= 2(W^TWx - x)(Wx)^T\end{aligned}$$

With respect to W:

$$\begin{aligned}\nabla_{W}F &= \delta^{(2)}a^{(1)T} \\ &= (W)(2(W^TWx - x))x^T\end{aligned}$$

Taking sums, noting that we need to transpose the gradient with respect to WT to get the gradient with respect to W, yields the final gradient with respect to W (pardon the slight abuse of notation here):

$$\begin{aligned}\nabla_{W}F &= \nabla_{W}F + (\nabla_{W^T}F)^T \\ &= (W)(2(W^TWx - x))x^T + 2(Wx)(W^TWx - x)^T\end{aligned}$$

Step 1b: Implement cost and gradient

In the file softICACost.m, implement the RICA cost and gradient. The cost we use is:

$$\min_{W} \quad \lambda \left\|Wx\right\|_1 + \frac{1}{2}\left\|W^TWx - x\right\|_2^2$$

Note that this is slightly different than the cost used in the gradient derivation section above (because we have added the L1 regularization and scaled the reconstruction term down by 0.5). To implement the L1-norm, we suggest using: $f(x) = \sqrt{x^2 + \epsilon}$ for some small ϵ. In this exercise, we find $\epsilon = 0.01$ to work well.

When done, check your gradient implementation. You could do this either using your own checkNumericalGradient.m from previous sections, or by using minFunc's built-in checker.

Self-Taught Learning

Self-Taught Learning

Overview

Assuming that we have a sufficiently powerful learning algorithm, one of the most reliable ways to get better performance is to give the algorithm more data. This has led to the that aphorism that in machine learning, "sometimes it's not who has the best algorithm that wins; it's who has the most data."

One can always try to get more labeled data, but this can be expensive. In particular, researchers have already gone to extraordinary lengths to use tools such as AMT (Amazon Mechanical Turk) to get large training sets. While having large numbers of people hand-label lots of data is probably a step forward compared to having large numbers of researchers hand-engineer features, it would be nice to do better. In particular, the promise of self-taught learning and unsupervised feature learning is that if we can get our algorithms to learn from unlabeled data, then we can easily obtain and learn from massive amounts of it. Even though a single unlabeled example is less informative than a single labeled example, if we can get tons of the former---for example, by downloading random unlabeled images/audio clips/text documents off the internet---and if our algorithms can exploit this unlabeled data effectively, then we might be able to achieve better performance than the massive hand-engineering and massive hand-labeling approaches.

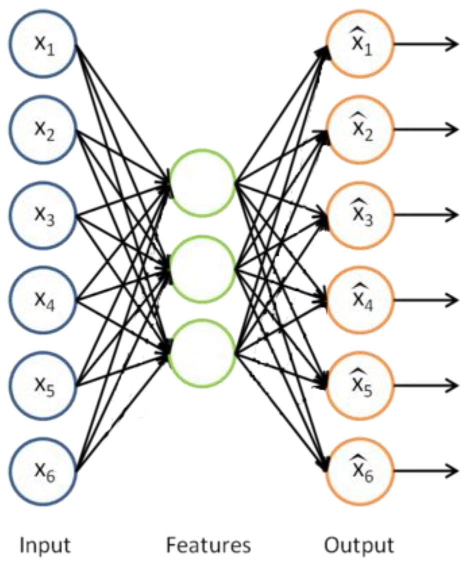
In Self-taught learning and Unsupervised feature learning, we will give our algorithms a large amount of unlabeled data with which to learn a good feature representation of the input. If we are trying to solve a specific classification task, then we take this learned feature representation and whatever (perhaps small amount of) labeled data we have for that classification task, and apply supervised learning on that labeled data to solve the classification task.

These ideas probably have the most powerful effects in problems where we have a lot of unlabeled data, and a smaller amount of labeled data. However, they typically give good results even if we have only labeled data (in which case we usually perform the feature learning step using the labeled data, but ignoring the labels).

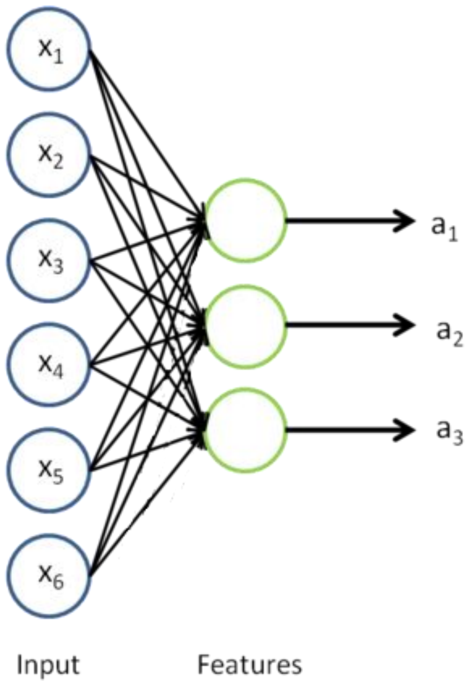
Learning features

We have already seen how RICA can be used to learn features from unlabeled data. Concretely, suppose we have an unlabeled training set $\{x_u^{(1)}, x_u^{(2)}, \ldots, x_u^{(m_u)}\}$

with m_u unlabeled examples. (The subscript "u" stands for "unlabeled.") We can then train an RICA on this data (perhaps with appropriate whitening or other pre-processing):



Having trained the parameters $W^{(1)}$ of this model, given any new input x, we can now compute the corresponding vector of activations a of the hidden units. As we saw previously, this often gives a better representation of the input than the original raw input x . We can also visualize the algorithm for computing the features/activations a as the following neural network:



This is just the RICA that we previously had, with with the final layer removed.

Now, suppose we have a labeled training set

$\{(x_l^{(1)}, y^{(1)}), (x_l^{(2)}, y^{(2)}), \ldots (x_l^{(m_l)}, y^{(m_l)})\}$ of m_l examples. (The subscript "l" stands for "labeled.") We can now find a better representation for the inputs. In particular, rather than representing the first training example as $x_l^{(1)}$, we can feed $x_l^{(1)}$ as the input to our RICA, and obtain the corresponding vector of activations $a_l^{(1)}$. To represent this example, we can either just replace the original feature vector with $a_l^{(1)}$. Alternatively, we can concatenate the two feature vectors together, getting a representation $(x_l^{(1)}, a_l^{(1)})$.

Thus, our training set now becomes $\{(a_l^{(1)}, y^{(1)}), (a_l^{(2)}, y^{(2)}), \ldots (a_l^{(m_l)}, y^{(m_l)})\}$ (if we use the replacement representation, and use $a_l^{(i)}$ to represent the i -th training example), or $\{((x_l^{(1)}, a_l^{(1)}), y^{(1)}), ((x_l^{(2)}, a_l^{(1)}), y^{(2)}), \ldots, ((x_l^{(m_l)}, a_l^{(1)}), y^{(m_l)})\}$ (if we use the concatenated representation). In practice, the concatenated representation often works better; but for memory or computation representations, we will sometimes use the replacement representation as well.

Finally, we can train a supervised learning algorithm such as an SVM, logistic regression, etc. to obtain a function that makes predictions on the y values. Given a test example $x_{\rm test}$, we would then follow the same procedure: For feed it to RICA to get $a_{\rm test}$. Then, feed either $a_{\rm test}$ or $(x_{\rm test}, a_{\rm test})$ to the trained classifier to get a prediction.

On pre-processing the data

During the feature learning stage where we were learning from the unlabeled training set $\{x_u^{(1)}, x_u^{(2)}, \ldots, x_u^{(m_u)}\}$, we may have computed various pre-processing parameters. For example, one may have computed a mean value of the data and subtracted off this mean to perform mean normalization, or used PCA to compute a matrix U to represent the data as $U^T x$ (or used PCA whitening or ZCA whitening).

If this is the case, then it is important to save away these preprocessing parameters, and to use the same parameters during the labeled training phase and the test phase, so as to make sure we are always transforming the data the same way to feed into RICA. In particular, if we have computed a matrix U using the unlabeled data and PCA, we should keep the same matrix U and use it to preprocess the labeled examples and the test data. We should not re-estimate a different U matrix (or data mean for mean normalization, etc.) using the labeled training set, since that might result in a dramatically different pre-processing transformation, which would make the input distribution to RICA very different from what it was actually trained on.

On the terminology of unsupervised feature learning

There are two common unsupervised feature learning settings, depending on what type of unlabeled data you have. The more general and powerful setting is the self-taught learning setting, which does not assume that your unlabeled data xu has to be drawn from the same distribution as your labeled data xl. The more restrictive setting where the unlabeled data comes from exactly the same distribution as the labeled data is sometimes called the semi-supervised learning setting. This distinctions is best explained with an example, which we now give.

Suppose your goal is a computer vision task where you'd like to distinguish between images of cars and images of motorcycles; so, each labeled example in your training set is either an image of a car or an image of a motorcycle. Where can we get lots of unlabeled data? The easiest way would be to obtain some random collection of images, perhaps downloaded off the internet. We could then train RICA on this large collection of images, and obtain useful features from them. Because here the unlabeled data is drawn from a different distribution than the labeled data (i.e., perhaps some of our unlabeled images may contain cars/motorcycles, but not every image downloaded is either a car or a motorcycle), we call this self-taught learning.

In contrast, if we happen to have lots of unlabeled images lying around that are all images of either a car or a motorcycle, but where the data is just missing its label (so you don't know which ones are cars, and which ones are motorcycles), then we could use this form of unlabeled data to learn the features. This setting---where each unlabeled example is drawn from the same distribution as your labeled examples---is sometimes called the semi-supervised setting. In practice, we often do not have this sort of unlabeled data (where would you get a database of images where every image is either a car or a motorcycle, but just missing its label?), and so in the context of learning features from unlabeled data, the self-taught learning setting is more broadly applicable.

Exercise: Self-Taught Learning

Overview

In this exercise, we will use the self-taught learning paradigm with convolutional nerual network, RICA and softmax classifier to build a classifier for handwritten digits.

You will be building upon your code from the earlier exercises. First, you will train your RICA on patches extracted from an "unlabeled" training dataset of handwritten digits. This produces filters that are penstroke-like. We then extract features from a labeled dataset of handwritten digits by convolving with these learnt filters. These features will then be used as inputs to the softmax classifier that you wrote in the previous exercise.

Concretely, for each example in the the labeled training dataset x_l, we forward propagate the example through a convolutional and a pooling layer to obtain the activation of the hidden units $a^{(2)}$. We now represent this example using $a^{(2)}$ (the "replacement" representation), and use this to as the new feature representation with which to train the softmax classifier.

Finally, we also extract the same features from the test data to obtain predictions.

In this exercise, our goal is to distinguish between the digits from 0 to 4. We will use an "unlabeled" dataset with all 10 digits to learn the filters; we will then use a labeled dataset with the digits 0 to 4 with which to train the softmax classifier.

In the starter code, we have provided a file stlExercise.m that will help walk you through the steps in this exercise.

Dependencies

The following additional files are required for this exercise:

MNIST Dataset

Support functions for loading MNIST in Matlab

The stl folder of the exercises starter code

You will also need your code from the following exercises:

RICA

Softmax Regression

Convolutional Neural Network

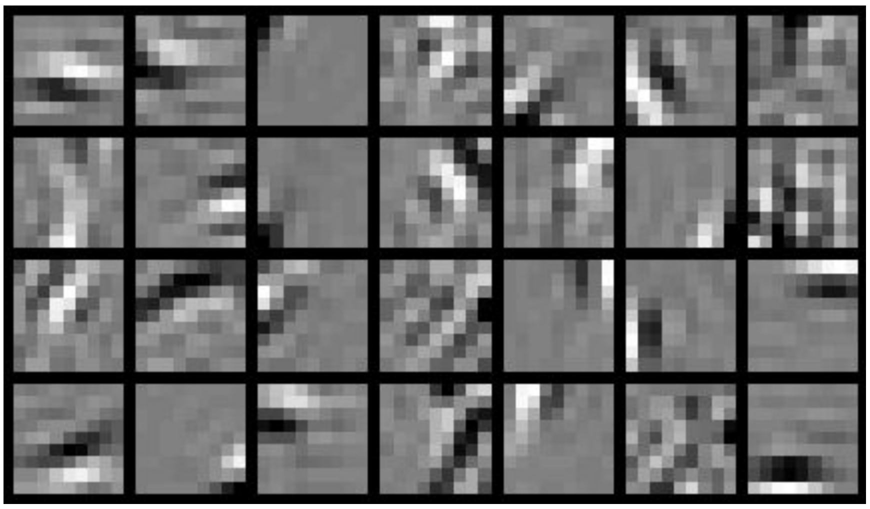
If you have not completed the exercises listed above, we strongly suggest you complete them first.

Step 1: Generate the input and test data sets

Go to the stl folder of the exercises code, which contains starter code for this exercise. Additionally, you will need to download the datasets from the MNIST Handwritten Digit Database for this project.

Step 2: Train RICA

In the starter code, we have provided code to split the MNIST dataset into 50000 "unlabelled" images and 10000 "labelled" images. We also provide code to randomly extract 200000 8-by-8 patches from the unlabelled dataset. You will need to whiten the patches using thezca2.m function seen in the RICA exercise. Then you will train an RICA on the 200000 patches, using the same softICACost.m function as you had written in the previous exercise. (From the earlier exercise, you should have a working and vectorized implementation of the RICA.) For us, the training step took less than 25 minutes on a fast desktop. When training is complete, you should get a visualization of pen strokes like the image shown below:



Informally, the features learned by the RICA should correspond to edge detectors.

Step 3: Extracting features

After the RICA is trained, you will use it to extract features from the labelled handwritten digit images. To extract features from an image of hand-written digit, you will first convolve the learnt RICA weights with the image, followed by RICA-style square-square-root pooling on the response.

Complete feedForwardRICA.m to produce a matrix whose columns correspond to activations of the hidden layer for each example, i.e., the vector a(2) corresponding to activation of layer 2. (Recall that we treat the inputs as layer 1).

Step 4: Training and testing the softmax regression model

Use your code from the softmax exercise (softmax_regression_vec.m) to train a softmax classifier using the training set features (trainFeatures) and labels (trainLabels).

Step 5: Classifying on the test set

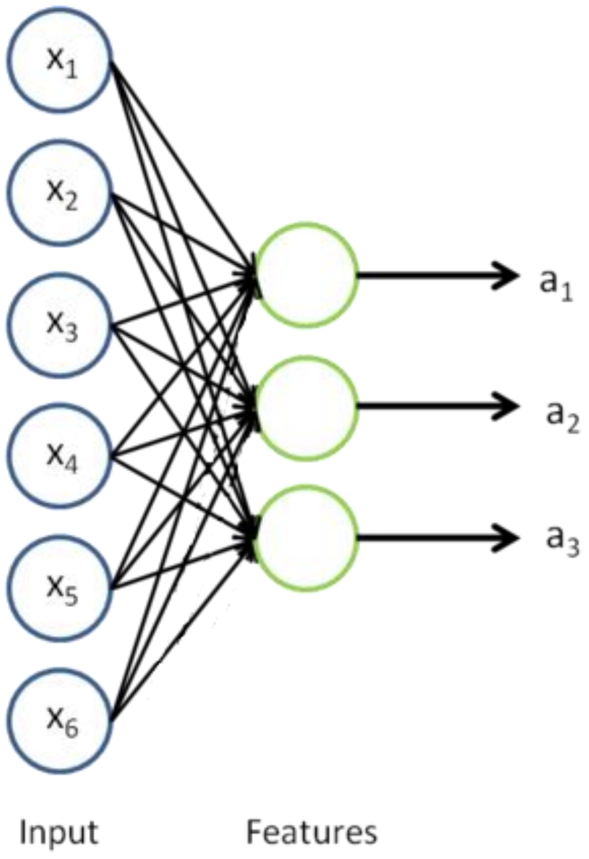
Finally, complete the code to make predictions on the test set (testFeatures) and see how your learned features perform! If you've done all the steps correctly, you should get 100% train accuracy and ~99% test accuracy. As a comparison, we get 97.5% test accuracy with random convolutional weights. Actual results may vary as a result of random initializations

Building Deep Networks for Classification

From Self-Taught Learning to Deep Networks

In the previous section, you used an autoencoder to learn features that were then fed as input to a softmax or logistic regression classifier. In that method, the features were learned using only unlabeled data. In this section, we describe how you can fine-tune and further improve the learned features using labeled data. When you have a large amount of labeled training data, this can significantly improve your classifier's performance.

In self-taught learning, we first trained a sparse autoencoder on the unlabeled data. Then, given a new example x, we used the hidden layer to extract features a. This is illustrated in the following diagram:



We are interested in solving a classification task, where our goal is to predict labels y. We have a labeled training set $\{(x_l^{(1)}, y^{(1)}), (x_l^{(2)}, y^{(2)}), \ldots (x_l^{(m_l)}, y^{(m_l)})\}$ of m_l labeled examples. We showed previously that we can replace the original features $x^{(i)}$ with features $a^{(l)}$ computed by the sparse autoencoder (the "replacement" representation). This gives us a training set

$\{(a^{(1)}, y^{(1)}), \ldots (a^{(m_l)}, y^{(m_l)})\}$. Finally, we train a logistic classifier to map from the features $a^{(i)}$ to the classification label $y^{(i)}$. To illustrate this step, similar to our earlier notes, we can draw our logistic regression unit (shown in orange) as follows:

380px

Now, consider the overall classifier (i.e., the input-output mapping) that we have learned using this method. In particular, let us examine the function that our classifier uses to map from from a new test example x to a new prediction p(y = 1 | x). We can draw a representation of this function by putting together the two pictures from above. In particular, the final classifier looks like this:

500px

The parameters of this model were trained in two stages: The first layer of weights $W^{(1)}$ mapping from the input x to the hidden unit activations a were trained as part of the sparse autoencoder training process. The second layer of weights $W^{(2)}$ mapping from the activations a to the output y was trained using logistic regression (or softmax regression).

But the form of our overall/final classifier is clearly just a whole big neural network. So, having trained up an initial set of parameters for our model (training the first layer using an autoencoder, and the second layer via logistic/softmax regression), we can further modify all the parameters in our model to try to further reduce the training error. In particular, we can fine-tune the parameters, meaning perform gradient descent (or use L-BFGS) from the current setting of the parameters to try to reduce the training error on our labeled training set $\{(x_l^{(1)}, y^{(1)}), (x_l^{(2)}, y^{(2)}), \ldots (x_l^{(m_l)}, y^{(m_l)})\}$.

When fine-tuning is used, sometimes the original unsupervised feature learning steps (i.e., training the autoencoder and the logistic classifier) are called pre-training. The effect of fine-tuning is that the labeled data can be used to modify the weights W(1) as well, so that adjustments can be made to the features a extracted by the layer of hidden units.

So far, we have described this process assuming that you used the "replacement" representation, where the training examples seen by the logistic classifier are of the form (a(i),y(i)), rather than the "concatenation" representation, where the examples are of the form((x(i),a(i)),y(i)). It is also possible to perform fine-tuning too using the "concatenation" representation. (This corresponds to a neural network where the input units xi also feed directly to the logistic classifier in the output layer. You can draw this using a slightly different type of neural network diagram than the ones we have seen so far; in particular, you would have edges that go directly from the first layer input nodes to the third layer output node, "skipping over" the hidden layer.) However, so long as we are using finetuning, usually the "concatenation" representation has little advantage over the "replacement" representation. Thus, if we are using fine-tuning usually we will do so with a network built using the replacement representation. (If you are not using fine-tuning however, then sometimes the concatenation representation can give much better performance.)

When should we use fine-tuning? It is typically used only if you have a large labeled training set; in this setting, fine-tuning can significantly improve the performance of your classifier. However, if you have a large unlabeled dataset (for unsupervised feature learning/pre-training) and only a relatively small labeled training set, then fine-tuning is significantly less likely to help.

Deep Networks: Overview

Overview

In the previous sections, you constructed a 3-layer neural network comprising an input, hidden and output layer. While fairly effective for MNIST, this 3-layer model is a fairly shallow network; by this, we mean that the features (hidden layer activations a(2)) are computed using only "one layer" of computation (the hidden layer).

In this section, we begin to discuss deep neural networks, meaning ones in which we have multiple hidden layers; this will allow us to compute much more complex features of the input. Because each hidden layer computes a non-linear transformation of the previous layer, a deep network can have significantly greater representational power (i.e., can learn significantly more complex functions) than a shallow one.

Note that when training a deep network, it is important to use a non-linear activation function $f(\cdot)$ in each hidden layer. This is because multiple layers of linear functions would itself compute only a linear function of the input (i.e., composing multiple linear functions together results in just another linear function), and thus be no more expressive than using just a single layer of hidden units.

Advantages of deep networks

Why do we want to use a deep network? The primary advantage is that it can compactly represent a significantly larger set of fuctions than shallow networks. Formally, one can show that there are functions which a k-layer network can represent compactly (with a number of hidden units that is polynomial in the number of inputs), that a (k − 1)-layer network cannot represent unless it has an exponentially large number of hidden units.

To take a simple example, consider building a boolean circuit/network to compute the parity (or XOR) of n input bits. Suppose each node in the network can compute either the logical OR of its inputs (or the OR of the negation of the inputs), or compute the logical AND. If we have a network with only one input, one hidden, and one output layer, the parity function would require a number of nodes that is exponential in the input size n. If however we are allowed a deeper network, then the network/circuit size can be only polynomial in n.

By using a deep network, in the case of images, one can also start to learn part-whole decompositions. For example, the first layer might learn to group together pixels in an image in order to detect edges (as seen in the earlier exercises). The second layer might then group together edges to detect longer contours, or perhaps detect simple "parts of objects." An even deeper layer might then group together these contours or detect even more complex features.

Finally, cortical computations (in the brain) also have multiple layers of processing. For example, visual images are processed in multiple stages by the brain, by cortical area "V1", followed by cortical area "V2" (a different part of the brain), and so on.

Difficulty of training deep architectures

While the theoretical benefits of deep networks in terms of their compactness and expressive power have been appreciated for many decades, until recently researchers had little success training deep architectures.

The main learning algorithm that researchers were using was to randomly initialize the weights of a deep network, and then train it using a labeled training set $\{(x_l^{(1)}, y^{(1)}), \ldots, (x_l^{(m_l)}, y^{(m_l)})\}$ using a supervised learning objective, for example by applying gradient descent to try to drive down the training error. However, this usually did not work well. There were several reasons for this.

Availability of data

With the method described above, one relies only on labeled data for training. However, labeled data is often scarce, and thus for many problems it is difficult to get enough examples to fit the parameters of a complex model. For example, given the high degree of expressive power of deep networks, training on insufficient data would also result in overfitting.

Local optima

Training a shallow network (with 1 hidden layer) using supervised learning usually resulted in the parameters converging to reasonable values; but when we are training a deep network, this works much less well. In particular, training a neural network using supervised learning involves solving a highly non-convex optimization problem (say, minimizing the training error $\sum_i ||h_W(x^{(i)}) - y^{(i)}||^2$ as a function of the network parameters W). In a deep network, this problem turns out to be rife with bad local optima, and training with gradient descent (or methods like conjugate gradient and L-BFGS) no longer work well.

Diffusion of gradients

There is an additional technical reason, pertaining to the gradients becoming very small, that explains why gradient descent (and related algorithms like L-BFGS) do not work well on a deep networks with randomly initialized weights. Specifically, when using backpropagation to compute the derivatives, the gradients that are propagated backwards (from the output layer to the earlier layers of the network) rapidly diminish in magnitude as the depth of the network increases. As a result, the derivative of the overall cost with respect to the weights in the earlier layers is very small. Thus, when using gradient descent, the weights of the earlier layers change slowly, and the earlier layers fail to learn much. This problem is often called the "diffusion of gradients."

A closely related problem to the diffusion of gradients is that if the last few layers in a neural network have a large enough number of neurons, it may be possible for them to model the labeled data alone without the help of the earlier layers. Hence, training the entire network at once with all the layers randomly initialized ends up giving similar performance to training a shallow network (the last few layers) on corrupted input (the result of the processing done by the earlier layers).

Greedy layer-wise training

How can we train a deep network? One method that has seen some success is the greedy layer-wise training method. We describe this method in detail in later sections, but briefly, the main idea is to train the layers of the network one at a time, so that we first train a network with 1 hidden layer, and only after that is done, train a network with 2 hidden layers, and so on. At each step, we take the old network with k − 1 hidden layers, and add an additional k-th hidden layer (that takes as input the previous hidden layer k − 1that we had just trained). Training can either be supervised (say, with classification error as the objective function on each step), but more frequently it is unsupervised (as in an autoencoder; details to be provided later). The weights from training the layers individually are then used to initialize the weights in the final/overall deep network, and only then is the entire architecture "fine-tuned" (i.e., trained together to optimize the labeled training set error).

The success of greedy layer-wise training has been attributed to a number of factors:

Availability of data

While labeled data can be expensive to obtain, unlabeled data is cheap and plentiful. The promise of self-taught learning is that by exploiting the massive amount of unlabeled data, we can learn much better models. By using unlabeled data to learn a good initial value for the weights in all the layers $W^{(l)}$ (except for the final classification layer that maps to the outputs/predictions), our algorithm is able to learn and discover patterns from massively more amounts of data than purely supervised approaches. This often results in much better classifiers being learned.

Better local optima

After having trained the network on the unlabeled data, the weights are now starting at a better location in parameter space than if they had been randomly initialized. We can then further fine-tune the weights starting from this location. Empirically, it turns out that gradient descent from this location is much more likely to lead to a good local minimum, because the unlabeled data has already provided a significant amount of "prior" information about what patterns there are in the input data.

In the next section, we will describe the specific details of how to go about implementing greedy layer-wise training.

Stacked Autoencoders

Overview

The greedy layerwise approach for pretraining a deep network works by training each layer in turn. In this page, you will find out how autoencoders can be "stacked" in a greedy layerwise fashion for pretraining (initializing) the weights of a deep network.

A stacked autoencoder is a neural network consisting of multiple layers of sparse autoencoders in which the outputs of each layer is wired to the inputs of the successive layer. Formally, consider a stacked autoencoder with n layers. Using notation from the autoencoder section, let W(k,1),W(k,2),b(k,1),b(k,2) denote the parameters W(1),W(2),b(1),b(2) for kth autoencoder. Then the encoding step for the stacked autoencoder is given by running the encoding step of each layer in forward order:

$$a^{(l)} = f(z^{(l)})$$
$$z^{(l+1)} = W^{(l,1)}a^{(l)} + b^{(l,1)}$$

The decoding step is given by running the decoding stack of each autoencoder in reverse order:

$$a^{(n+l)} = f(z^{(n+l)})$$
$$z^{(n+l+1)} = W^{(n-l,2)}a^{(n+l)} + b^{(n-l,2)}$$

The information of interest is contained within a(n), which is the activation of the deepest layer of hidden units. This vector gives us a representation of the input in terms of higher-order features.

The features from the stacked autoencoder can be used for classification problems by feeding a(n) to a softmax classifier.

Training

A good way to obtain good parameters for a stacked autoencoder is to use greedy layer-wise training. To do this, first train the first layer on raw input to obtain parameters W(1,1),W(1,2),b(1,1),b(1,2). Use the first layer to transform the raw input into a vector consisting of activation of the hidden units, A. Train the second layer on this vector to obtain parameters W(2,1),W(2,2),b(2,1),b(2,2). Repeat for subsequent layers, using the output of each layer as input for the subsequent layer.

This method trains the parameters of each layer individually while freezing parameters for the remainder of the model. To produce better results, after this phase of training is complete, fine-tuning using backpropagation can be used to improve the results by tuning the parameters of all layers are changed at the same time.

Concrete example

To give a concrete example, suppose you wished to train a stacked autoencoder with 2 hidden layers for classification of MNIST digits, as you will be doing in the next exercise.

First, you would train a sparse autoencoder on the raw inputs x(k) to learn primary features h(1)(k) on the raw input.

400px

Next, you would feed the raw input into this trained sparse autoencoder, obtaining the primary feature activations h(1)(k) for each of the inputs x(k). You would then use these primary features as the "raw input" to another sparse autoencoder to learn secondary featuresh(2)(k) on these primary features.

400px

Following this, you would feed the primary features into the second sparse autoencoder to obtain the secondary feature activations h(2)(k) for each of the primary features h(1)(k) (which correspond to the primary features of the corresponding inputs x(k)). You would then treat these secondary features as "raw input" to a softmax classifier, training it to map secondary features to digit labels.

400px

Finally, you would combine all three layers together to form a stacked autoencoder with 2 hidden layers and a final softmax classifier layer capable of classifying the MNIST digits as desired.

500px

Discussion

A stacked autoencoder enjoys all the benefits of any deep network of greater expressive power.

Further, it often captures a useful "hierarchical grouping" or "part-whole decomposition" of the input. To see this, recall that an autoencoder tends to learn features that form a good representation of its input. The first layer of a stacked autoencoder tends to learn first-order features in the raw input (such as edges in an image). The second layer of a stacked autoencoder tends to learn second-order features corresponding to patterns in the appearance of first-order features (e.g., in terms of what edges tend to occur together--for example, to form contour or corner detectors). Higher layers of the stacked autoencoder tend to learn even higher-order features.

Fine-tuning Stacked AEs

Introduction

Fine tuning is a strategy that is commonly found in deep learning. As such, it can also be used to greatly improve the performance of a stacked autoencoder. From a high level perspective, fine tuning treats all layers of a stacked autoencoder as a single model, so that in one iteration, we are improving upon all the weights in the stacked autoencoder.

General Strategy

Fortunately, we already have all the tools necessary to implement fine tuning for stacked autoencoders! In order to compute the gradients for all the layers of the stacked autoencoder in each iteration, we use the Backpropagation Algorithm, as discussed in the sparse autoencoder section. As the backpropagation algorithm can be extended to apply for an arbitrary number of layers, we can actually use this algorithm on a stacked autoencoder of arbitrary depth.

Finetuning with Backpropagation

For your convenience, the summary of the backpropagation algorithm using element wise notation is below:

1. Perform a feedforward pass, computing the activations for layers L_2, L_3, up to the output layer L_{n_l}, using the equations defining the forward propagation steps.

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

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

(When using softmax regression, the softmax layer has $\nabla J = \theta^T (I - P)$ where I is the input labels and P is the vector of conditional probabilities.)

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. Compute the desired partial derivatives:

$$\nabla_{W^{(l)}} J(W,b;x,y) = \delta^{(l+1)} (a^{(l)})^T,$$
$$\nabla_{b^{(l)}} J(W,b;x,y) = \delta^{(l+1)}.$$
$$J(W,b) = \left[\frac{1}{m} \sum_{i=1}^m J(W,b;x^{(i)},y^{(i)}) \right]$$

Excercise:Implement Deep Network for Digit Classification

There is currently no text in this page. You can search for this page title in other pages, or search the related logs.