Robert On
05571627

Semi-Supervised Feature Learning with Neural Networks

Neural Networks and Dimension Reduction on Large, Sparse Feature Spaces

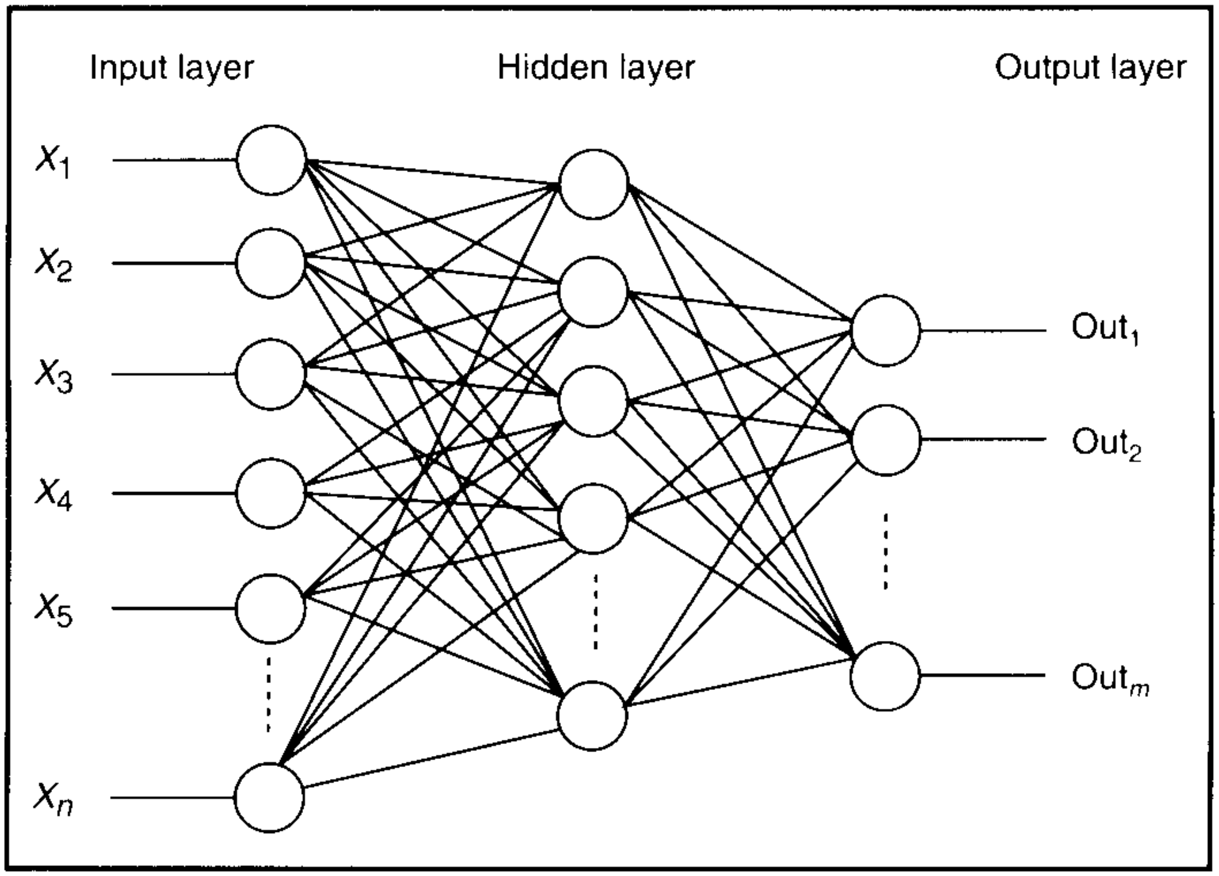
The primary goal of this project was to explore the challenge of semi-supervised feature learning on a relatively large, sparse feature space. A large body of work has already been well established in the field of supervised and unsupervised feature learning on relatively small feature spaces but the challenge with a relatively large feature space is not as well explored. The primary difficulty in dealing with large feature spaces are largely computational. Since computation for large feature spaces is both expensive and time consuming, exploring and evaluating the theoretical and practical efficacy of various learning algorithms can be slow to completely infeasible.

The specific challenged I attempted to tackle is the one specified by the following Kaggle competition: http://www.kaggle.com/c/SemiSupervisedFeatureLearning. The objective here is to learn a set of 100 "rich" features from a space of about a million features. Each example in the dataset typically only has a few activated (non-zero) features and had a binary valued (-1 or 1) output label. There were two training data sets available for this competition: 1) a large (1M) set of unlabeled training examples and 2) a small (50K) set of labeled training examples, in addition to a corresponding set of labeled test data.

The main goal here, as with many Kaggle competitions, was to do well on the test dataset. However there was a subgoal of producing a set of 100 features that would represent the high-dimensional feature set best. With the relatively large number of learning algorithms available to tackle this problem, I decided to focus on that of neural networks. Neural networks are a very powerful in their expressive power and generalizability. Furthermore, they can be reconfigured to tackle problems in a variety of ways and their hidden layers can provide strong insight and intuition on the importance. By assembling a neural network with 100 nodes in the final hidden layer, I would be able to learn a complex representation of the high-dimensional feature set while training the network to achieve the highest accuracy.

Initially I intended to approach the construction of the 100 features with a Deep Belief Network employing a stacked autoencoder using greedy layer-wise training since there was a significant amount (1M) of unlabeled data available. This quickly became theoretically and practically unattractive. With the availability of labeled data, the performance of supervised learning to learn meaningful features appeared much more straightforward than compressing the input feature space with an autoencoder and then learning from that. Computationally, the multi-layer autoencoder turned out to be computationally impractical with such a large input feature space as it meant computing with at least two matrices with dimensions of at least 1M x 100.

I ultimately decided to go simpler by assembling a simple 2-layer, logistic regression, neural network with 1M input features, 100 nodes in the hidden layer, and a two-node output layer. With less hidden layers and a smaller output layer, this became much more computationally feasible than the stacked autoencoder. Although simpler, this model was still able to come up with a set of 100 features from the hidden layer that would learn the best weights for approximating the labeled training data set. I decided to not use the unlabeled set at all.



I closely followed the neural network setup done in exercise four to load the new dataset, train the dataset with three values of the regularization parameter and 50 iterations of backpropagation. The 150 total iterations on the high-dimensional data took several hours to complete. The results were as follows:

lambda	*Cost (after 50 iterations)*	*Training Set Accuracy*	*Test Set Accuracy*
0.5	2.718292	*95.444*	*97.612*
1.0	4.989151	95.322	96.041
1.5	7.376167	95.420	97.256

The best training and test set accuracy resulted from training the network with the smallest (0.5) regularization parameter. We can interpret this to mean that the algorithm could benefit from fitting the training data more precisely, thereby trading off some variance for bias.

As mentioned earlier, the learned weights from a neural network can provide valuable insight on the features and their importance to predicting its outcomes. The first thing I observed was the sum of absolute weights coming from each of the input nodes to the hidden layer. The absolute weights here provide some insight into what the most weighted (or valuable) input features were. The top 100 features by this measure were in agreement across the three different values of lambda for 99 out of 100 features and there were approximately in the same order with approximately the same weights. The sorted indices of these features and their weights are included in the appendix.

The input weights to each of the nodes of the 100 node hidden layer assemble a complex combination of the input features to create 100 rich features that is then combined in the output node to make a prediction. Since each of these nodes have weights for the 1M input features, including them in this report is not practical but I saved this matrix to disk as a way to shrink the feature space of future input data. However, this part is crucial as it addresses the subgoal of finding a rich but reduced features space.

The weights of the hidden layer to the output layer provide some insight as to which of the features from the reduced space were important to the final prediction. The following are the top important features from the hidden layer for predicting a negative example in ascending order:

4 7 56 55 97 79 22 84 76 57

and a positive example in ascending order:

4 79 7 56 55 75 97 84 76 57

This reassures us that the significant features in predicting a positive and negative example are very similar.

Neural networks are an incredibly powerful and generalizable tool for machine learning.

However, its computational costs can be very limiting. Future work in the computational scalability of neural networks will hold great promise for machine learning and its applications.

Appendix

Top input weights:

```
# Created by Octave 3.2.3, Sat Dec 10 17:51:20 2011 PST <robon@gpu-
corp.mtv.corp.google.com>
# name: top100_weights
# type: matrix
# rows: 1
# columns: 101

2.694319670923756 2.696070198734571 2.698518361879555 2.699096588485051
2.699121245137382 2.699465804693998 2.700922633591048 2.702413462708362
2.704916370511591 2.706355556437934 2.709097767207371 2.71000000684436
2.711442953535624 2.711562812236154 2.71499883354452 2.716540476731025
2.717331376261127 2.718630730113211 2.719509193339201 2.72425483926272
2.726399619445517 2.728553026466911 2.728714273818349 2.728765425807596
2.730172714603496 2.731946613164157 2.735207687309947 2.737019699674836
2.743285322354333 2.744867568084593 2.745063678687089 2.746628523169271
2.753259689716664 2.754076319267781 2.756086110022512 2.761236314681125
2.761930602252748 2.765263095905118 2.773533925423982 2.773721750372681
2.779493447030254 2.780134301230607 2.781166850893808 2.782841632338162
2.786673658953007 2.787394134650617 2.789044913018446 2.789472472184352
2.792908371637689 2.795039150433938 2.799517404745654 2.800539269898939
2.802617546481971 2.810063968499595 2.810788900851614 2.81081771684875
2.811701499963145 2.816265653911542 2.819120269487535 2.820142797335007
2.825018012441294 2.828494831720668 2.83077128075348 2.830948882633178
2.846642957421374 2.847358951242132 2.851593713958705 2.854258338839572
2.863179771013413 2.866717002568106 2.866962628039834 2.871860657670975
2.880679457109681 2.88232708437456 2.891575586517805 2.898199979683255
2.913334111998936 2.913946541946857 2.932822638404757 2.933005881048456
2.947958443324982 2.969789528219365 2.981348677374422 2.988351409012465
3.006253162079331 3.007285385167471 3.041807009210191 3.044332701646297
3.046308365817673 3.048276828625017 3.056451632885363 3.118988900917754
3.142162379194288 3.154944283586611 3.158736598922706 3.167859913659215
3.182162020175638 3.886321756072266 4.169366451955416 8.3168089324299
8.484534761989224
# name: top100_index
# type: matrix
# rows: 1
# columns: 101
 710548 645519 572798 142885 549134 30637 971175 707895 871095 395617 821435 833813
941200 147192 870482 288964 810791 409888 801337 813543 473968 450054 238666 601990
385151 631763 742199 239352 314855 562313 538700 103748 490592 769775 100890 787374
473940 831597 202863 805666 686348 627720 711954 341858 548216 248325 450722 901409
586802 862859 793263 592341 912730 889718 193879 313720 277651 375058 910029 779467
449187 934618 420421 780832 625264 352876 475417 433528 520173 210436 402647 132130
738433 373657 619529 524412 888547 273733 554078 889885 578559 21408 300593 54593
749922 137504 860359 148339 782201 303857 949587 260967 243349 983360 452021 839689
212661 864909 112395 157745 16406
```

Code:

```
clear ; close all; clc

input_layer_size  = 999999;
hidden_layer_size = 100;
num_labels = 2;
```

```
fprintf('Loading Data ...\n')

% load training data
load('matlab_format/public_train_data.matlab.sparsematrix.dat');
load('matlab_format/public_train.labels.dat');
X = spconvert(public_train_data_matlab_sparsematrix);
clear('public_train_data_matlab_sparsematrix');
y = public_train_labels;
clear('public_train_labels');
y(y == 1) = 2;
y(y == -1) = 1;

% load test data
load('matlab_format/public_test.labels.dat');
load('matlab_format/public_test_data.matlab.sparsematrix.dat');
X_test = spconvert(public_test_data_matlab_sparsematrix);
clear('public_test_data_matlab_sparsematrix');
X_test = [X_test zeros(50000, 5)];
y_test = public_test_labels;
clear('public_test_labels');
y_test(y_test == 1) = 2;
y_test(y_test == -1) = 1;

m = size(X, 1);

fprintf('\nInitializing Neural Network Parameters ...\n')

initial_Theta1 = randInitializeWeights(input_layer_size, hidden_layer_size);
initial_Theta2 = randInitializeWeights(hidden_layer_size, num_labels);

initial_nn_params = [initial_Theta1(:) ; initial_Theta2(:)];

fprintf('\nTraining Neural Network... \n')

options = optimset('MaxIter', 50);

for lambda = [0.5 1 1.5]
  % Create "short hand" for the cost function to be minimized
  costFunction = @(p) nnCostFunction(p, ...
                                     input_layer_size, ...
                                     hidden_layer_size, ...
                                     num_labels, X, y, lambda);

  % Now, costFunction is a function that takes in only one argument (the
  % neural network parameters)
  [nn_params, cost] = fmincg(costFunction, initial_nn_params, options);

  % Obtain Theta1 and Theta2 back from nn_params
  Theta1 = reshape(nn_params(1:hidden_layer_size * (input_layer_size + 1)), ...
                   hidden_layer_size, (input_layer_size + 1));

  Theta2 = reshape(nn_params((1 + (hidden_layer_size * (input_layer_size + 1))
):end), ...
                   num_labels, (hidden_layer_size + 1));

  pred = predict(Theta1, Theta2, X);
  train_accuracy = mean(double(pred == y)) * 100;
  save(['train_accuracy_100_', num2str(lambda)], 'train_accuracy');
  fprintf('\nTraining Set Accuracy: %f\n', mean(double(pred == y)) * 100)

  pred = predict(Theta1, Theta2, X_test);
  test_accuracy = mean(double(pred == y_test)) * 100;
  save(['test_accuracy_100_', num2str(lambda)], 'test_accuracy');
  fprintf('\nTest Set Accuracy: %f\n', mean(double(pred == y_test)) * 100)

  feature_sums = sum(Theta1);
  [s, si] = sort(abs(feature_sums));
  top100_weights = s(1,999900:1000000);
  top100_index = si(1,999900:1000000);
  save(['top100_100_', num2str(lambda)], 'top100_weights', 'top100_index');

  save(['Theta2_100_', num2str(lambda)], 'Theta2');
  save(['Theta1_100_', num2str(lambda)], 'Theta1');
endfor
```