Further Details for Homework 2, CS 273A

Kevin Bache

October 10, 2015

This homework is similar to homework 1 in that you're going to be model accuracies for a variety of hyperparameter values. Where in the previous homework your models were some closed-form linear discriminant function and the perceptron, here your models will be nerual networks trained with backpropagation. Where in the previous homework, the hyperparameters to vary were $\alpha, N, N_{iterations}$, and N_{test}, here the parameters to vary are learning rate, number of backpropagation iterations and optionally the strength of the weight decay term.

Let's dive into a few details from the homework.

- **"using back-propagation of error (online or batch)"** – Here "batch" refers to "batch gradient descent" and "online" refers to "online gradient descent" or "stochastic gradient descent"[1]. Batch gradient descent is the normal "gradient descent" algorithm which you might have heard of before.

  Let's back up for a second and set the stage. Say that you had a model with a bunch of parameters (for example your W matrices and biases) and you squished all of those parameters into one big vector called θ. Say that we also have a loss function called $L(\theta)$ which furthermore, can be expressed as a weighted sum of per-datapoint losses: $L_{total}(\theta) = \frac{1}{N}\sum_{i=1}^{N} L_i(\theta)$. Here, $L_i(\theta)$ represents the loss for datapoint i.

  We wish to minimize this loss function and one way to do that is with gradient descent. With a loss function that can be broken up into a sum over datapoint-specific loss values, the derivative with respect to element j in the θ vector will be: $\frac{\partial L(\theta)}{\partial \theta_j} = \frac{\partial \frac{1}{N}\sum_{i=1}^{N} L_i(\theta)}{\partial \theta_j} = \frac{1}{N}\sum_{i=1}^{N} \frac{\partial L_i(\theta)}{\partial \theta_j}$.

  You can then define a small, scalar learning rate of, say, $\alpha \in [0.00001, 0.1]$, and iteratively update your θ vector with the gradient descent update $\theta \leftarrow \theta - \alpha \frac{\mathrm{d}L(\theta)}{\mathrm{d}\theta}$. The negative sign is because the gradient will point up hill on the loss surface and we wish to move down hill to a minima.

  The problem with this batch gradient descent scheme is that just performing one calculation of the gradient will take $\mathcal{O}(ND)$ time, and if N, the number of datapoints is large (e.g.: $1e9$) then this can get prohibitively expensive. And remember, this is just the cost of performing one gradient descent iteration, and we might have to do a *lot* of iterations of our gradient descent update (say, 1,000 to 1,000,000) before our neural net converges.

  Stochastic gradient descent (SGD) is a simple, yet incredibly powerful solution to the problem of calculating expensive gradients. Where in batch gradient descent we calculate the gradient of the overall loss function as the average of the gradient estimates from all N datapoints, in SGD we simply estimate the gradient as the average of the gradient estimates from a smaller "minibatch" of B randomly selected datapoints: $\widehat{\frac{\partial L(\theta)}{\partial \theta_j}} = \frac{1}{B}\sum_{i \in \text{Minibatch}-\text{IDs}} \frac{\partial L_i(\theta)}{\partial \theta_j}$. Note that I've added a hat to the partial derivative definition on the left to indicate that this isn't the true partial derivative with respect to θ_j, but rather an estimate of that value. Now, our SGD update is just $\theta \leftarrow \theta - \alpha \widehat{\frac{\mathrm{d}L(\theta)}{\mathrm{d}\theta}}$

[1] In truth there are some subtl differences between "online" and "stochastic" gradient descent but people often use them interchangeably in practice and for our porposes they're the same

This stochastic gradient is a noisier estimate of the true uphill direction in our loss surface than the actual gradient, but it's expected value is the same and it's much, much faster to compute[2]. B values tend to range between 1 and around 1000, with 30 - 100 being most common for large datasets[3].

- **Weight decay**. Weight decay is a regularization strategy which means that it attempts to ensure that a model which you've learned on the training data will generalize well to unseen data. The idea of weight decay is to modify your loss function to penalize you for having large parameter values. There are lots of ways to perform weight decay, but "L2 weight decay" is probably the most common. If $L(\theta)$ represents your loss function calculated for a particular of your parameter vecotr θ, then L2 weight decay modifies your loss function by adding in a second term that penalizes you for having large weights: $L'(\theta) = L(\theta) + \frac{\lambda}{2}\theta^T\theta$. Here λ represents the strength of the weight decay term and $L'(\theta)$ represents the modified loss function which can be optimized similarly to the unmodified loss function: simply take repeated steps in the direction of the negative gradient (negative since the gradient points up hill on our loss function and we wish to minimize our loss). Good λ values vary from problem to problem, but a reasonable first guess might be something in the range of 0.0001 to 0.1.

- **Autoencoder**. An autoencoder is a neural network in which the output that you're trying to predict is the full input vector. This would be trivially easy if you had more hidden units in your hidden layer than you have input units, so to make it do something useful, you make the number of hidden units smaller than the number of input and output units. For example, if you had a 100 dimensional input vector, you might have 10 hidden units in your first hidden layer and then expand out again to 100 output units. This forces the autoencoder to learn to compress your data. The output won't usually be perfect, but if trained well it can learn to reasonably approximate your input data.

  You might generate your data with something like this. Feel free to fiddle with $\mathrm{n}_data, n_dim_full, andn_dim_limitedtomakeyo$

```
from __future__ import division
# if you're using python 2.7, the division operator, '/' performs integer
# division by default so for instance 1 / 4 ==> 0.  to prevent this, the
# first line of your programs should always be
# "from __future__ import division" so that 1 / 4 ==> 0.25
# also, __future__ imports must be the first lines in your program.
# they're weird.

# you don't have to rename numpy to np, but it's a very common standard
# for numerical python code and one that's good to follow
import numpy as np

# always set your random seed in so that you can repeat your results
np.random.seed(1234)

# main idea here:
# we're going to sample a bunch of data from a multivariate normal
```

[2]And in fact there are some very good reasons to prefer SGD over batch gradient descent for optimizing machine learning problems. In short, batch gradient descent optimizes loss on your training dataset, while stochastic gradient descent optimizes expected loss on unseen data. For more details, see section three of this excellent paper by one of the masterminds behind modern SGD theory: http://research.microsoft.com/pubs/192769/tricks-2012.pdf

[3]Bigger minibatches are actually more computationally efficient than small minibatches. To see why, take the extreme case of $B = 1$. Now your neural network will need to multiply your one datapoint x_i by your weight matrices to perform a forward pass through your neural network. With a larger minibatch of, say, $B = 100$, you're now multiplying a matrix, $X_{\text{minibatch}}$ which as 100 rows, by the weight matrices. Matrix-matrix multiplication can be performed much more efficiently than a series of matrix-vector multiplications, so larger minibatches lead to faster computation per datapoint.

This gain in computational efficiency, however, needs to be balanced against a need for statistical efficiency. As you add more and more datapoints to a minibatch, the quality of your gradient estimate will plateau, and you won't get any extra benefit from adding additional data to your minibatch.

The trade-off between these two factors, the computational efficiency of large minibatches vs. the redundancy of the extra datapoints in large ones is why minibatch sizes of 32-128 tend to be the norm

```
# distribution, but we're going to set up it's covariance matrix so that
# the data won't fill the full space, but instead will live in a
# lower-dimensional subspace within it
#
# you don't need to understand the details of how this works, but it
# might help to understand what the effect is. for example, if you set
# n_dim_full = 3 and n_dim_limited = 2, your sampled datapoints (the
# rows of the x matrix below) will more or less span a 2-dimensional
# pancake in 3-dimensional space.  the same holds for any arbitrary
# dimensions that you set in n_dim_full and n_dim_limited

# the number of dimension of the full space
n_dim_full = 100
# the dimensionality of the subspace in which the data will more-or-less
# live
n_dim_limited = 30
# the number of datapoints
n_data = 10000

# make a diagonal matrix to serve as the eigenvalues of the covariance
# matrix that we'll make in a second.  it will have n_dim_full big
# eigenvalues (which will lead to dimensions with high variance) and
# n_dim_limited small eigenvalues (dimensions with low variance)
eigenvals_big = np.random.randn(n_dim_limited) + 3
eigenvals_small = np.abs(np.random.randn(n_dim_full - n_dim_limited)) * .1
eigenvals = np.concatenate([eigenvals_big, eigenvals_small])
print eigenvals
diag = np.diag(eigenvals)

# construct q, an arbitrary orthonormal matrix
q, r = np.linalg.qr(np.random.randn(n_dim_full, n_dim_full))
# construct a covariance matrix as Q * D * Qt.  this is essentially a
# singular value decomposition done in reverse.  this method of
# constructing the covariance matrix ensures that it's positive
# semidefinite (which it must be to be a valid covariance matrix) and
# also lets us set its eigenvalues which helps us ensure that the data
# we'll sample from a multivariate normal using this covariance matrix
# lives in a n_dim_limited dimensional subspace of the full n_dim_full
# space.
cov_mat = q.dot(diag).dot(q.T)

# the mean vector of the covariance matrix
mu = np.zeros(n_dim_full)

# sample some data
x = np.random.multivariate_normal(mu, cov_mat, n_data)
```

- **some logic function on binary vectors such as exactly-one-on**. You might try something like this but with more data and more dimensions:

```
from __future__ import division
import numpy as np

np.random.seed(10)

# number of datapoints
```

```
n_data = 15
# number of features or dimensions
n_dim = 6

# random values in [0, 1) that are smaller than this will be set to 1 in the
# x_matrix, 0 otherwise
p_threshold = 2 / n_dim

# matrix of random values in the range [0, 1)
p_matrix = np.random.rand(n_data, n_dim)

# binarize this matrix so that values bigger than p_threshold will become 1
# and the rest become 0
x_matrix = 1 * (p_matrix < p_threshold)

# find rows which have exactly one element turned on
y = 1 * (np.sum(x_matrix, 1) == 1)

# helper function for pretty printing
def print_named(thing_to_print, name):
    header_bar = '=' * (len(name) + 1)
    print header_bar
    print name + ':'
    print header_bar
    print thing_to_print
    print

print_named(p_matrix, 'p_matrix')
print_named(p_threshold, 'p_threshold')
print_named(x_matrix, 'x_matrix')
print_named(y, 'y')
```

- **Show a visual representation (similar to a heat map) of the final trained weights for your best network**. Each layer in your neural net contains a matrix of weights. One way to visualize the final trained weights of the network might be to show the matrix as an image, with one pixel representing each weight and the color of the pixel indicating how strongly negative or positive that weight ended up as. Another option might be to show a histogram of the weights at each layer.

- **Probability distributions to use in reporting a single average error measurement includes a distribution (eg a narrow Gaussian around zero) over starting weight arrays, and a distribution on input patterns from which you draw your training and test sets; just say what distributions you chose.** You're going to end up with a table similar to the one from homework 1 in which each cell of the table tells you the mean and standard deviation of your training and test error. Think for a second about where you get each of the accuracy values that contributes to that mean and standard deviation. One way to get these different accuracy values might be to randomly generate different datasets, and report your model's accuracy on each of these (which is what you did in the previous homework). Another way might be to keep a single dataset and instead to vary the initial randomly assigned weights of your network. Here, Prof. Mjolsness is saying that you should specify how you got your different accuracy values – by varying the starting weights of your network or the dataset that you used to train your network.