PROJECT 3: GENERIC BOLTZMANN MACHINES AND RESTRICTED BOLTZMANN MACHINES

NABAR, ADITI

1. Database

This assignment had two objectives. The first was to build a generic Boltzmann machine and observe it's stabilization by monitoring the absolute differences of the means from consecutive iterations. The second objective was to build a Restricted Boltzmann Machine RBM, train a set of weights for the reconstruction of some given input data, and using those weights to initialize a multi-layer perceptron, perform a classification task.

1.1. **Define Features and Classes.** Each image from the MNIST dataset is made of 28 by 28 or 784 pixels. The set of features will be defined as $F = \{F_1, F_2, \ldots, F_{784}\}$ where each F_p corresponds to a single pixel in a given image. Each F_p will store a value between 0 and 1 representing the amount of gray intensity. The higher or lower the number indicates the intensity of dark or light color respectively. The classes are defined by $C_1, C_2, \ldots, C_r, \ldots, C_N$ where N is the number of input data points, and the value assigned to each class C_r is the label of image r. Though the classes are being presented in this section, they are only relevant to the latter portion of the project.

For the restricted Boltzmann machine, the RBM, the machine is non-deterministic. It's goal is not to classify the input data into a set of classes. The job of the restricted boltzman machine is to learn and reconstruct the input data, with high probability. This obviously dictates the defining of the features as specified above. It also means that the size of the visible units n_1 will be the same as the size of the reconstruction n_3, so that $n_1 = n_3$.

1.2. **Define Training and Test Sets.** We are using the MNIST dataset which consists of 70,000 images of hand written numbers split into 60,000 points of training data, and 5,000 points of test data.

2. Software Tools

The data set is loaded with the help of *python-mnist*. *Python-mnist* parses the MNIST dataset and returns two lists: the training set and the test set. Both sets contained a list of images and list of labels. *Scikit-learn* is used to analyze the performance of the model, namely with the help of the *pca*

Date: April 6, 2017.

module. Finally, I used the module *matplotlib* for plotting the results of my analyses.

With computing the reconstruction and root mean-squared-error (RMSE) per batch, I was unable to run the batch-wise RMSE for the RBM on my local machine to completion in a timely manner. Thus I am unable to show an assessment of the RMSE across batches.

3. General Boltzmann Machines

A Boltzmann machine is a network of nodes that determine the state of the nodes x_i in a stochastic manner, where x_i is defined as:

$$x_i = \begin{cases} +1 & on \\ -1 & off \end{cases}$$

Upon presenting the machine with a binary vector representation of the data, the machine must then learn to reconstruct that binary representation. The machine accomplishes this by finding weights on certain connections so that the data vectors yield a low cost compared to non-data binary vectors. The cost is measured in terms of the energy of the configuration of the machine.

The Boltzmann machine stochastic dynamics are as follows: - The Boltzmann machine iteratively and randomly steps through the set of nodes. When the machine reaches each unit i, it computes a total value v_i for each site.

$$v = \sum_i W_{ij} \cdot v_i$$

where $W_{ij} = W_{ji} \neq 0$.

The logistic sigmoid function will then use this value to compute a probability that after the visit to the node, the the node will take the On state.

$$\mathbf{P}(y_i = +1) = \frac{1}{1 + e^{-v_i}}$$

As long as the units continue to be updated in a stochastic manner, one in which the visitation is memory-less, then after a few full sweeps, the machine will achieve a stabilization of stochastic behavior that approximates a Boltzmann distribution $G(x)$. In other words, $\mathbf{P}(X(t) = x) \simeq G(x)$. A Boltzmann distribution determines the probability of a configuration based on the energy of that configuration, relative to other non-data binary vectors.

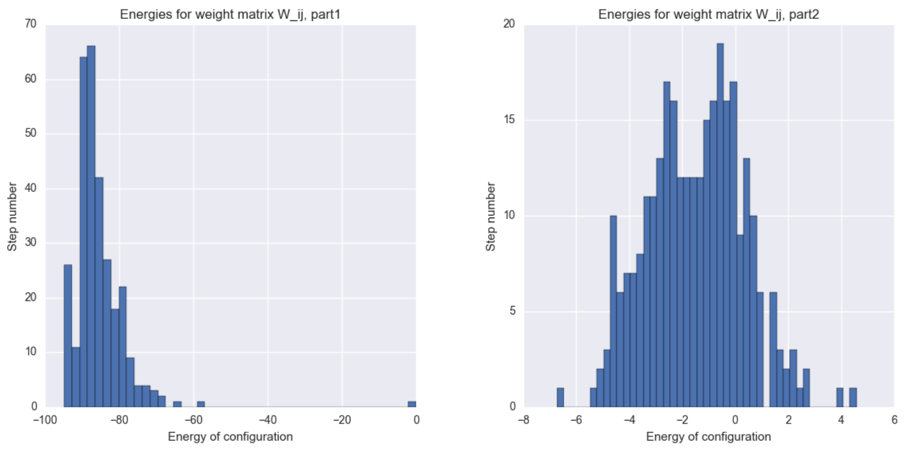
$$\mathbf{P}(y) = \frac{e^{-E(v)}}{\sum_u e^{-E(u)}}$$

Regardless of whether or not the machine approximates a Boltzmann distribution, the energy of a configuration is given by

$$E(y) = \sum_i x_i^{\mathrm{y}} \cdot b_i - \sum_{i<j} x_i^{\mathrm{y}} \cdot x_j^{\mathrm{y}} \cdot w_{ij}$$

.

The Boltzmann machine was run with two sets of initial weights. The first set of initial weights W_1 were randomly sampled from the interval $[-1, 1]$, and the second set of weights was set to $W_2 = W_1/10$. Define stabilization as the time (or sweep) $s(j)$ such that the absolute value of the difference of consecutive empirical means of unit j, or $|M_{s+t+1}(j) - M_{s+t}(j)|$, remains below a set threshold for 10 consecutive sweeps. Given a threshold of 0.01, the general Boltzmann machine built for this project was observed to stabilize at sweep 171 for W_1 and at sweep 271 for W_2. The histograms below demonstrate the differences in the quantiles of energy for the two weights. The energy histogram for W_1 is right-skewed, while the energy histogram for W_2 appears to be normally distributed with insignificant skew.



4. Fast RBMs

The algorithm used for the fast restricted Boltzmann machine is as follows. The input data is broken into batches, which are fed into the machine as visible units $visible_1$. These visible units are then passed into the logistic sigmoid function along with their weights and biases, producing the hidden units $hidden_1$. This computes the probabilities $\mathbf{P}(y_i = +1)$ for the hidden nodes. Then $hidden_1$ is used to compute the probabilities for reconstruction, or $visible_2$. $visible_2$ is then used to compute the probabilities for the hidden nodes $hidden_2$. $hidden_2$ is eventually used for the contrastive divergence method used to update the weights, but $visible_2$ is the desired reconstruction, after epochs of training.

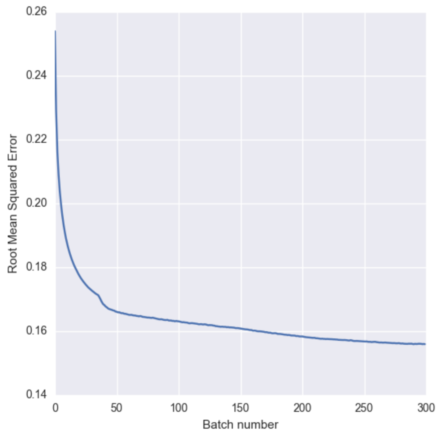
4.0.1. *Root Mean Squared Error.* The goal of a restricted Boltzmann machine is to learn from the data, and with high accuracy, be able to reconstruct the input data itself. The root mean squared error (RMSE) can be computed to analyze how closely the reconstruction matches the data it is approximating, the input data. There are a number of steps involved before computing the root mean squared error. First, the hidden cloud must be explicitly computed by passing the images in the batch X_{batch}, and their associated weights W, and biases b_h, through the sigmoid activation function.

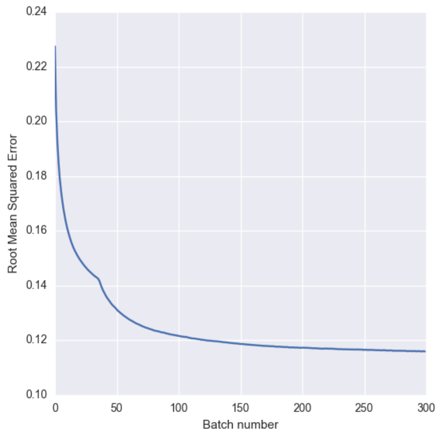
$$\sigma_{hidden} = \frac{1}{1 + e^{-(W \cdot X_{batch} + b_h)}}$$

The hidden cloud then is passed back through the sigmoid function, with the transpose of the weights W, and the bias of the visible units b_v to obtain a reconstruction of the input data *recon*. This reconstruction is then passed into the root mean squared error to provide a RMSE per batch.

$$\sigma_{recon} = \frac{1}{1 + e^{-(W^T \cdot \sigma_{hidden} + b_v)}}$$

The restricted Boltzmann machine was run for 70 epochs, and the RMSE was computed at the end of every 20th batch, for both the training set and the test set.





The RMSE(W*) for the entire training data was found to be 1.01050, and the corresponding observed RMSE for the whole test data was 0.78802. It is interesting that the RMSE for the test data was found to be lower than that of the training data.

4.0.2. *Options for Initializing Weights.* The following are options presented by *Keras* for initializing the weights used by a neural network. For the purposes of this project, the weight options were not that significant, as we were tasked with initializing the first layer of the multi-layer perceptron with weights trained by the restricted Boltzmann machine. The second layer however was initialized by sampling from the normal distribution.

Keras allows for the initialization of weights at the time the neural network is initialized. Options for initialization include:

- *uniform*: sampling from the Uniform distribution
- *lecun_uniform*: the Uniform multiplied by the square root of the number of inputs as per Lecun 1998
- *normal*: sampling from the Normal distribution

- *identity*: generating as an identity matrix
- *orthogonal*: generating as a random orthogonal matrix
- *zeros*: generating a matrix of 0's
- *ones*: generating a matrix of 1's
- *Glorot* initializations: Initialization using the Glorot method scaled by $fan_{in} + fan_{out}$. With W as the sampling distribution, fan_{in} as the number of inputs a hidden unit(a) receives, and fan_{out} as the number of units the output of unit a is sent to, Glorot's initialization samples from the Normal (*glorot_normal*) or Uniform distributions (*glorot_uniform*), with $\mu = 0$ and σ given by

$$\text{Var}(W) = \frac{2}{fan_{in} + fan_{out}}$$

- *He* initializations: Initialization using the He method scaled by fan_{in}. With W as the sampling distribution and fan_{in} as the number of inputs a hidden unit (a) receives, the He initialization methods, like the Glorot, sample from the Normal (*he_normal*) and the Uniform (*he_uniform*) distributions with $\mu = 0$ and σ given by

$$\text{Var}(W) = \frac{2}{fan_{in}}$$

4.0.3. *Options for Batch Learning.* Using *keras* makes it very easy to train a neural network. The *model.fit* function is called using a *batch_size* parameter. When the parameter is given a value, the model is trained in batches. In the absence of the parameter, the model is processed as one large batch.

4.0.4. *Options for Stopping Learning. Keras* has an *EarlyStopping* function in its *Callbacks()* class that will stop training when a specified quantity has stopped changing. This function allows for the monitoring of the following

- accuracy
- loss
- loss if validation is enabled during training
- accuracy if the validation is enabled during training

allowing you to stop with specified conditions. *Keras* allows for customizable parameters for monitoring. This is done by making a new class and inheriting from *keras.callbacks.Callbacks()* and writing custom values. The values can be gathered at the beginning of each batch, the end of a batch, or the end of an epoch.

4.0.5. *Software Output.* We are able print the following performance measures: accuracy and loss. These measures can be printed per batch and per epoch. Any other output is done after training.

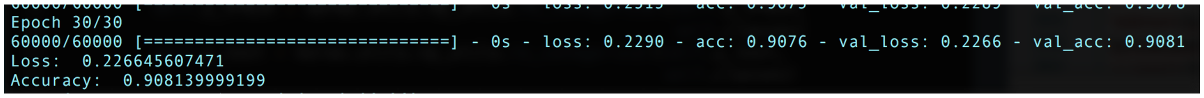
5. PCA Analysis

I conducted a principal component analysis (PCA) on the RBM on each hidden layer, given the training set and the test set. A principal component analysis isolates the n components, or in this case, the n learned features, that explain the variance in the data. For this PCA, I set $n = 3$ principal components. The number of components needed to achieve 90% of the energy in the model were observed as follows:

Data	Hidden Units	Number of Components
training	32	23
test	32	22
training	64	>32
test	64	>32

6. Autoencoder Efficiency

Using the weights trained for reconstruction by the restricted Boltzmann machine, I ran a multi-layer perceptron to classify the images, and to assess the performance of the RBM. After running 30 epochs, the MLP achieved a final loss of 0.2290, from an initial loss of 0.3232. The accuracy of the MLP did not change significantly, increasing slightly from 0.9000 to 0.9076. Since validation was enabled during training, the loss given validation and accuracy given validation are also relevant, and were observed to stop at 0.2266 and 0.9081 respectively.



```
Epoch 30/30
60000/60000 [==============================] - 0s - loss: 0.2290 - acc: 0.9076 - val_loss: 0.2266 - val_acc: 0.9081
Loss:  0.226645607471
Accuracy:  0.908139999199
```

References

[1] Chollet, François: Keras,
`https://github.com/fchollet/keras`

[2] NeuPy.com
`https://neupy.com/2015/09/20/discrete_hopfield_network.html`

[3] Scholarpedia.org
`http://www.scholarpedia.org/article/Boltzmann_machine#The_stochastic_dynamics _of_a_Boltzmann_machine`

[4] DeepLearning.net
`http://deeplearning.net/tutorial/rbm.html`