Journal of Machine Learning Research 20 (2019) 1-10 Submitted 11/19; Published 12/19

Machine Learning Extra Credit: Design Document

Brandon Sladek BRANDONSLADEK@GMAIL.COM
School of Computing
Montana State University
Bozeman, MT 59718, USA

Jared Thompson J.A.THOMPSON22@GMAIL.COM
School of Computing
Montana State University
Bozeman, MT 59718, USA

Kyle Hagerman HAGERMANKYLE96@GMAIL.COM
School of Computing
Montana State University
Bozeman, MT 59718, USA

Ryan Hansen RYANHANSEN2222@GMAIL.COM
School of Computing
Montana State University
Bozeman, MT 59718, USA

Abstract

In this paper we outline the design of a Python program that implements a stacked autoencoder and uses it to find meaningful encodings of the input data that can then be reliably used to perform classification. We also outline our experimental approach that we will use to test the stacked autoencoder implementation using six different data sets from the UCI machine learning repository. We will compare the various networks based on their prediction accuracy and average convergence rates using various hyperparameter values.

Keywords: Stacked Autoencoder, Neural Network, Gradient Descent

1. Introduction

Stacked autoencoders are a special type of neural network in which the input layer is the same size as the output layer. In the context of a stacked autoencoder, the goal of the network is to learn a compressed (encoded) representation of the input vector that can then be decoded to form the output layer in order to recover the original input vector. In other words, we want the network to learn a configuration of weights and biases such that the output matches the input as closely as possible, while forming a "bottleneck" by compressing the input in the middle layers. In real world applications, stacked autoencoders are most often used as a dimensionality reduction technique used to form a more compact input vector that preserves the most important features of the original input vector (Vincent et al. (2010)). In this way, stacked autoencoders can be said to "learn" the structure of the data, which can then be used as a reduced data set in other applications.

2. Design Overview

Our goal for this project is to compare the performance of a stacked autoencoder on various data sets. In this section we present an overview of our proposed architecture for a Python application that implements a stacked autoencoder, and stacks an additional neural network on top of the autoencoder to perform classification or regression using the code layer (compressed data representation) generated by the trained stacked autoencoder. The following document outlines the details of how we will implement, train, test, and tune the networks, and then compare their performances across the different data sets using various values for the hyperparameters.

2.1 Class Descriptions

The core of our design is based upon a main driver class, ExperimentRunner. This class will instantiate a class for each neural network implementation that extends a parent class, NeuralNetwork. The ExperimentRunner will also use the DataAPI, Preprocessor, and CrossValidator classes to call their respective methods. Specifically, the DataAPI class handles creating the Pandas DataFrames from the csv files. See Figure 2 at the end of the document for the associated UML diagram.

The Preprocessor class will call a general method that analyzes each raw DataFrame and handles replacing or imputing missing values. We also normalize all data at this stage and handle whether a dataset is to be used for classification (nominal target values) or regression (continuous target values).

The CrossValidator class will implement a 10-fold cross validation dividing each preprocessed data frame into ten subsets. In each iteration of the program, nine subsets will make up the training set and the tenth will be reserved as the test set. The test set will rotate each iteration so that each subset of the data will be used as a test set.

The ParameterTuner class will simply be used as a single source of parameters for each of the algorithms, fetched by calling the get_parameters() method and specifying the name of the algorithm for which we want the corresponding list of parameters to use in our experiment. It's a one-stop shop for any global definitions we need to make for the entire set of experiments we may wish to run.

The Results class will analyze each model's predictions post training using 0/1 Loss. It will print/save the results for later analysis to be presented in the final report.

Finally, the StackedAutoEncoder class will be configurable based on which network we are instantiating. The number of nodes in each of the layers will be arbitrarily determined with a constructor parameter.

2.2 Parameter Tuning

1. **Stacked Autoencoder**

   The layers in the stacked autoencoder will be trained using unsupervised pre-training.

2. **Accompanying Neural Network**

   The accompanying neural network will be trained using backpropagation, reusing the Multilayer Perceptron neural network implementation from project 3.

2.3 Major Decisions

1. **StackedAutoEncoder Class**

   Since a stacked autoencoder is essentially just a slight variation on regular neural networks, the StackedAutoEncoder class will extend the base NeuralNetwork class implemented in project 3. This way it will be able to reuse the corresponding backward_prop() and forward_prop() methods for training and testing.

2. **Loss Functions** - We care about two components of performance.
   Correctness - First is related to how well the machine is trained. We will gauge this by 0-1 loss, translated into an accuracy when divided by number of total attempts. Mean squared error for regression data. Note, this is based on output from the prediction layer. As long as we use the same prediction neural net for each auto encoder, we can attribute the difference in error to the auto encoder.
   Convergence - Second is related to runtime. How long does it take for the algorithm to finish training. We will be comparing these directly. A higher runtime for training means it took longer for the learning algorithm to converge.

3. **Activation Functions** For simplicity, we chose to use some of the most commonly used activation functions. For all non-output layers we will use the rectified linear unit (ReLU) activation function, and the sigmoid activation function for output layers. We will use ReLU for the non-output layers because it is the most commonly used activation function in modern deep learning. And we will use the sigmoid for the output layers so the values of our output nodes can readily be interpreted as probabilities when doing classification.

4. **Hidden Layer ("Code" Layer)**

   The "code" layer size (number of nodes) will be determined through the hyperparameter tuning process. Since the typical use case for autoencoders is dimensionality reduction, we will seek the smallest code layer size possible while still preserving the autoencoders ability to recover the input and classify using the code layer as a compressed feature vector.

5. **Backpropagation Objective Function** In order to do gradient descent, we need to try to optimize some objective function. We chose squared error, as it is widely used and simple to implement.

6. **Batching**

   We will use mini batching to decrease the time required for training the networks.

3. Experiment

This section discusses the experimental motivation and general approach for accepting or rejecting our hypotheses for the various data sets and parameter values.

3.1 Approach

We want to test the various stacked autoencoders on correctness and runtime. We are particularly interested in SAE with 1, 2, or 3 hidden layers. For implementation, we will rely heavily on the Pandas library (McKinney (2010)). The SAE will then be trained using unsupervised pre-training, and the network on top with backpropagation. Last, for the sake of consistency, we plan to use 10-fold cross validation to ensure our results are not particular to a specific training/test set.

3.2 Hypothesis

The following table contains our prediction for performance.

Algorithm	Accuracy Rank	Convergence Time Rank
SAE 1 Hidden Layer	1 - The data is 'compressed' less. We are doing less generalization	1 - Least HL
SAE 2 Hidden Layer	2 - More compressed than 1, less compressed than 3	2 - Medium HL
SAE 3 Hidden Layer	3 - Most compressesd. Patterns in 3rd HL are least fitted to data.	3 - Most HL

Figure 1: Table depicting our performance hypothesis. Justification is included.

4. Proposed Analysis

As mentioned in previous sections, our performance analysis is twofold. First, we gauge a form of correctness through 0-1 loss (accuracy) based on results from classification tests or mean squared error for regression. A higher percentage correct indicates a better performance. Second, we test convergence rate. We compare this based on runtime of each algorithm. A higher runtime is a worse convergence rate, and we will compare times directly.

References

Wes McKinney. Data structures for statistical computing in python, 2010. URL `https://pandas.pydata.org/`.

Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of machine learning research*, 11(Dec):3371–3408, 2010.

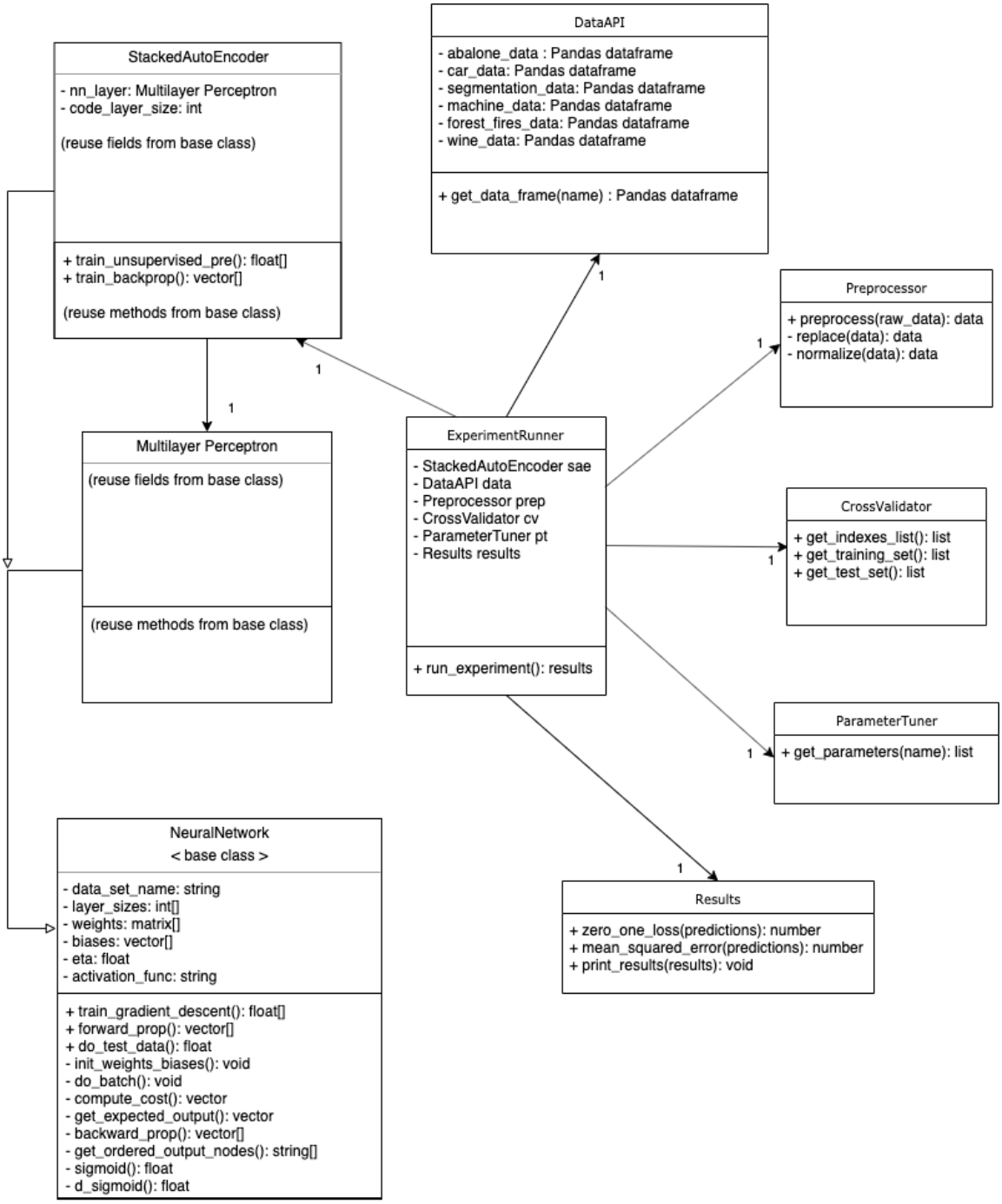

Figure 2: UML implementation of stacked auto encoder (with optional MLP network on top for prediction) and experiment suite for testing and comparison of results