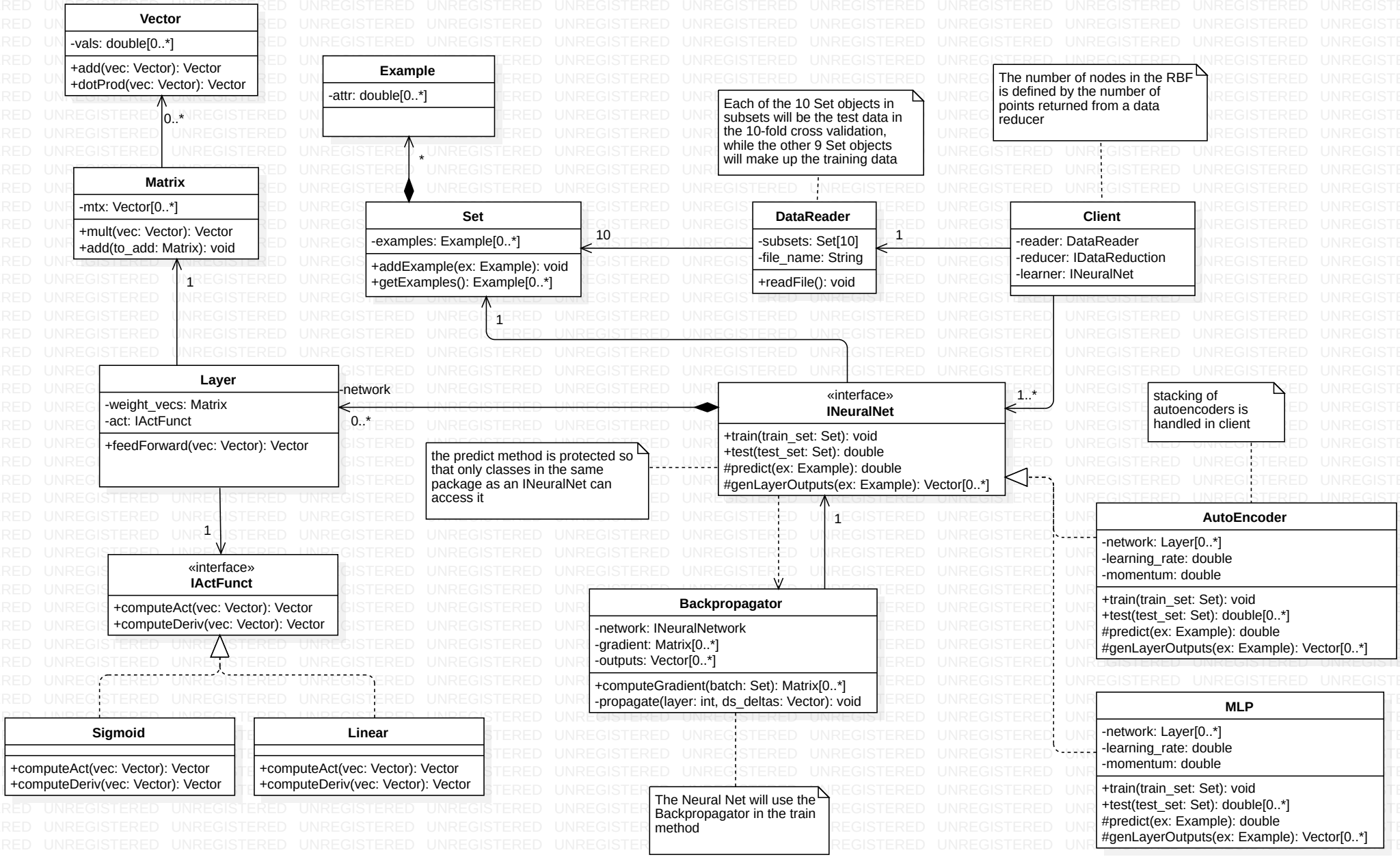
Extra Credit Design Document

CSCI 447
10/25/2019

Andrew Kirby
Kevin Browder
Nathan Stouffer
Eric Kempf


Vector
-vals: double[0..*]
+add(vec: Vector): Vector
+dotProd(vec: Vector): Vector
0..*
Example
-attr: double[0..*]
*
Each of the 10 Set objects in subsets will be the test data in the 10-fold cross validation, while the other 9 Set objects will make up the training data
The number of nodes in the RBF is defined by the number of points returned from a data reducer
Matrix
-mtx: Vector[0..*]
+mult(vec: Vector): Vector
+add(to_add: Matrix): void
1
Set
-examples: Example[0..*]
+addExample(ex: Example): void
+getExamples(): Example[0..*]
10
DataReader
-subsets: Set[10]
-file_name: String
+readFile(): void
1
Client
-reader: DataReader
-reducer: IDataReduction
-learner: INeuralNet
1
Layer
-weight_vecs: Matrix
-act: IActFunct
+feedForward(vec: Vector): Vector
-network
0..*
«interface»
INeuralNet
+train(train_set: Set): void
+test(test_set: Set): double
#predict(ex: Example): double
#genLayerOutputs(ex: Example): Vector[0..*]
1..*
stacking of autoencoders is handled in client
the predict method is protected so that only classes in the same package as an INeuralNet can access it
1
«interface»
IActFunct
+computeAct(vec: Vector): Vector
+computeDeriv(vec: Vector): Vector
Backpropagator
-network: INeuralNetwork
-gradient: Matrix[0..*]
-outputs: Vector[0..*]
+computeGradient(batch: Set): Matrix[0..*]
-propagate(layer: int, ds_deltas: Vector): void
AutoEncoder
-network: Layer[0..*]
-learning_rate: double
-momentum: double
+train(train_set: Set): void
+test(test_set: Set): double[0..*]
#predict(ex: Example): double
#genLayerOutputs(ex: Example): Vector[0..*]
Sigmoid
+computeAct(vec: Vector): Vector
+computeDeriv(vec: Vector): Vector
Linear
+computeAct(vec: Vector): Vector
+computeDeriv(vec: Vector): Vector
MLP
-network: Layer[0..*]
-learning_rate: double
-momentum: double
+train(train_set: Set): void
+test(test_set: Set): double[0..*]
#predict(ex: Example): double
#genLayerOutputs(ex: Example): Vector[0..*]
The Neural Net will use the Backpropagator in the train method

Class Description

DataReader: This class takes in a file name and reads in the data to instances of Set classes. The data is assumed to be in a standard format that a DataReader object can process.
Set: A Set is composed of a group of examples (data points). This will be used to store examples so that our learning algorithm can be trained and tested.
INeuralNet: This interface defines the methods required to implement a Neural Network. There are two types of networks that will be implemented: *Autoencoder* and *Multi-Layer Perceptron (MLP)*. Each have their own class that will handle the specifics of each type of network.
Layer: The Layer class composes networks. Each layer consists of a weight matrix and an activation function. An input vector can be passed into a layer to produce the output to the next layer.
IActFunct: This interface defines the methods used in an activation function. There are two types: *Sigmoidal* and *Linear*. Each take in a Vector and compute the activation of each component in the vector.
Backpropagator: The Backpropagator class is the weight learning algorithm. A network is passed in along with a batch to train on. The Backpropagator class then applies the backpropagation algorithm to compute the gradient of each weight in the network.

Algorithm Design

Stacked autoencoders have a repetitive structure and training process. Each stacked autoencoder may have an arbitrary number of autoencoder layers. The entire autoencoder is composed of the encoder—the layers which bottleneck the input to a lower number of nodes—and the decoder—the layers which attempt to reconstruct the input from output of the last encoder layer. Thus, the stacked autoencoder will always be built one encoder at a time, with each autoencoder following the initial being placed at the junction of the encoder and decoder. In other words, the first autoencoder is trained by inputting to the input layer and observing the output at the output layer. Each subsequent autoencoder is placed so that its input is the output of the deepest autoencoder's encoding layer and its output layer outputs to the same autoencoder's decoding layer. Additionally, each autoencoder has less nodes than the previous to provide the correct functionality. Weights of each autoencoder are locked after pretraining until all autoencoders have been stacked. Training of autoencoders trains layer weights so that inputs are maximally recovered at the output.

Once the stacked autoencoder is satisfactory, the encoding layers (bottleneck) may be returned and used as the first layers in a feed forward network. Further layers will be specified during construction of the feedforward network. With the feedforward network fused with the

autoencoder, a classification or regression output layer will allow for class or real value prediction. The feedforward prediction layers will then be trained using backpropagation (stopping at the last autoencoding layer). Lastly, the entire network will be trained using backpropagation.

The functionality of layers—nodes, edge weights, and backpropagation—are generic. Each layer may utilize linear or sigmoidal activation functions. The sigmoidal activation function of choice will be the logistic function. Backpropagation will function the same as in project three. All networks are implemented using the Strategy Pattern (since they implement the same interface). The stacked autoencoder will use linear activation functions in the hidden and output layers, as this is a very common configuration.. The feedforward network will be represented, built, and trained identical to project three.

Fine tuning of the stacked autoencoder and training of the feed forward network on gradient descent, which is implemented in the Backpropagator class. This class takes in the current weights as well as a training batch. For each example in the training batch, the algorithm will compute a prediction using the current weights. The gradient with respect to an individual weight is computed as $\frac{\partial Err}{\partial w_{ji}} = \delta_j * x_{ji}$ where x_{ji} is the i^{th} input to the j^{th} node and δ_j is the error term that is propagated backwards from the error at the output layer. The error function used at the output layer is Mean Squared Error. For hidden layers, δ_j is a function of δ_k (where δ_k represents the δ values in the next layer). This sets up a recursive structure for backpropagation that can be applied to any number of hidden layers. The gradient for each example in the batch is then averaged. This is returned to the learner for updates to the network.

Experimental Design

Validation

The network will be tested on every data set. Regression and Classification will be used as needed. Each test will involve ten-fold cross validation, which requires separate training and test sets that will be allocated during preprocessing.

Tuning

There is a lot of tuning that can be done on a stacked autoencoder and on the feed forward network. Starting with the stacked autoencoder we will tune the number of hidden nodes in each layer, learning rate and momentum. To tune the number of hidden nodes we will start with the number of nodes used in an overcomplete autoencoder and then tune up and down from that value to find an optimal value. The learning rate and momentum will be tuned the same as the feed forward network which is described below.

When tuning the feed forward network, the most important attribute to tune will be the number of nodes in each hidden layer. The optimum number of nodes in a hidden layer is somewhere between the number of nodes in the input and output layers (Heaton 2008). Tuning will start with the mean of the number of nodes in the input and output layers for classification problems and half the number of nodes in the input layer for regression. The momentum will also be tuned. Tuning of momentum will start at .5 as it is a common value to start tuning with (Goodfellow 2016). From the starting value we will tune downwards to find a more optimal local minimum. Lastly the learning rate will be tuned. Since the input data is normalized between 0 and 1, learning rates should be between 10^-6 and 1 and the starting value will be .1 because it is a standard starting value (Bengio 2012). We will tune in both directions from this starting value to find an optimal resolution and speed.

Evaluation Metrics

Since the autoencoder and feed forward network are trained separately we will need to use evaluation metrics on both. For classification problems on the feed forward network Mean Squared Error (MSE), and accuracy will be used. Classes do not have real value associated with them, so MSE is calculated as the square root of the sum of squared differences between total actual examples in a class and total predicted examples in a given class. In this case, MSE is a good indicator of how far off the predicted distribution is from the actual distribution. Accuracy indicates how well the algorithm is classifying examples on an individual basis. These two metrics supplement each other because it is possible to get a MSE of 0 and classify nothing right. Using both metrics give a better overall evaluation of an algorithms performance.

Mean Absolute Error (MAE) and MSE will be used to evaluate the regression problems and for stacked autoencoders the MAE and MSE will be taken for each attribute and then averaged across the entire network to get a metric for total performance of the stacked autoencoder. MSE takes the distance between real and predicted values and squares it. MAE is similar but takes the absolute value of the distance between real and predicted values. One issue with MSE is that if the error is less than 1 the performance is underestimated and if the error is greater than 1 the performance is overestimated. This can be addressed by using MAE as well because it is linear and will show if the MSE is over or underestimating the performance of the algorithm. An issue encountered in previous projects is that both MAE and MSE are proportional to the real-valued outputs. This means evaluating these metrics requires that the context (the scale of the outputs) is known. This will be accounted for in this project by computing the MAE and MSE of the z-scores. The mean and standard deviations used to compute the z-score will be computed based on the testing set.

References

Bengio, Y., 2012. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade* (pp. 437-478). Springer, Berlin, Heidelberg.

Goodfellow, I., Bengio, Y. and Courville, A., 2016. *Deep learning*. MIT press.

Jeff Heaton. 2008. Introduction to Neural Networks for Java, 2nd Edition (2nd ed.). Heaton Research, Inc..

Shichao Zhang, Xuelong Li, Ming Zong, Xiaofeng Zhu, and Debo Cheng. 2017. "Learning *k* for kNN Classification." ACM Trans. Intell. Syst. Technol. 8, 3, Article 43 (January 2017), 19 pages.

Svozil, D., Kvasnicka, V., Pospichal, J. "Introduction to multi-layer feed-forward neural networks." Chemometrics and Intelligent Laboratory Systems, Volume 39, Issue 1, 1997, Pages 43-62.