Automated Feature Engineering for Deep Neural Networks with Genetic Programming

by

Jeff Heaton

A dissertation proposal submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in
Computer Science

College of Engineering and Computing
Nova Southeastern University

2016

An Abstract of a Dissertation Proposal Submitted to Nova Southeastern University in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

Automated Feature Engineering for Deep Neural Networks with Genetic Programming

by
Jeff Heaton
2016

Feature engineering is a process that augments the feature vector of a predictive model with calculated values that are designed to enhance the accuracy of the model's predictions. Research has shown that the accuracy of models such as deep neural networks, support vector machines, and tree/forest-based algorithms sometimes benefit from feature engineering. Expressions that combine one or more of the original features usually create these engineered features. The choice of the exact structure of an engineered feature is dependent on the type of machine learning model in use. Previous research demonstrated that various model families benefit from different types of engineered feature. Random forests, gradient boosting machines, or other tree-based models might not see the same accuracy gain that an engineered feature allowed neural networks, generalized linear models, or other dot-product based models to achieve on the same data set.

The proposed dissertation seeks to create a genetic programming-based algorithm to automatically engineer features that might increase the accuracy of deep neural networks. For a genetic programming algorithm to be effective, it must prioritize the search space and efficiently evaluate what it finds. The algorithm will face a search space composed of all possible expressions of the original feature vector and evaluate candidate-engineered features found in the search space. Five experiments will provide guidance on how to prioritize the search space and how to most efficiently evaluate a potential engineered feature. Thus, the algorithm will have a greater opportunity to find engineered features that increase the predictive accuracy of the neural network. Finally, a sixth experiment tests the accuracy improvement of neural networks on data sets when features engineered by the proposed algorithm are added.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This dissertation proposal seeks to create an algorithm that will automatically engineer features that might increase the accuracy of deep neural networks for certain types of predictive problems. The proposed research builds upon, but does not duplicate, prior published research by the author of this dissertation. In 2008, the Encog Machine Learning Framework was created and includes advanced neural network and genetic programming algorithms (Heaton, 2015). The Encog genetic programming framework introduced an innovative algorithm that allows dynamically generated constant nodes for tree-based genetic programming. As a result, constants in Encog genetic programs can assume any value, rather than choosing from a fixed constant pool.

Research was performed that demonstrated the types of manually engineered features most likely to increase the accuracy of several models (Heaton, 2016). The research presented here builds upon this earlier research by leveraging the Encog genetic programming framework as a key component of the proposed algorithm that will automatically engineer features for a feedforward neural network that might contain many layers. This type of neural network is commonly referred to as a deep neural network (DNN). Although it would be possible to perform this research with any customizable genetic programming framework or deep neural network framework, Encog is well suited for the task because it provides both components.

This dissertation proposal begins by introducing both neural networks and feature engineering. The dissertation problem statement is defined, and a clear goal is established. Building upon this goal, the relevance of this study is demonstrated and includes a discussion of the barriers and issues previously encountered by the scientific community. A brief review of literature will show how this research continues previous investigations of deep learning. In addition to the necessary resources and the methods, the research approach to achieve the dissertation goal is outlined.

Most machine learning models, such as neural networks, support vector machines (Smola & Vapnik, 1997), and tree-based models, accept a vector of input data and then output a prediction based on this input. For these models, the inputs are called features, and the complete set of inputs is called a feature vector. Most business applications of neural networks must map the input neurons to columns in a database; these inputs allow the neural network to make a prediction. For example, an insurance company might use columns for age, income, height, weight, high-density lipoprotein (HDL) cholesterol, low-density lipoprotein (LDL) cholesterol, and triglyceride level (TGL) to make suggestions about an insurance applicant (B. F. Brown, 1998).

Inputs such as HDL, LDL, and TGL are all named quantities. This can be contrasted to high-dimensional inputs such as pixels, audio samples, and some time-series data. For consistency, this dissertation will refer to lower-dimensional data set features that have specific names as *named features*. This dissertation will center upon such named features. High-dimensional inputs that do not assign specific meaning to individual features fall outside the scope of this research.

Classification and regression are the two most common applications of neural networks. Regression networks predict a number, whereas classification networks assign a non-numeric class. For example, the maximum policy face amount is the maximum amount that the regression neural network suggests for an individual. This is a dollar amount, such as $100,000. Similarly, a classification neural network can suggest the non-numeric underwriting class for an individual, such as preferred, standard, substandard, or decline. Figure 1 shows both of these neural networks.

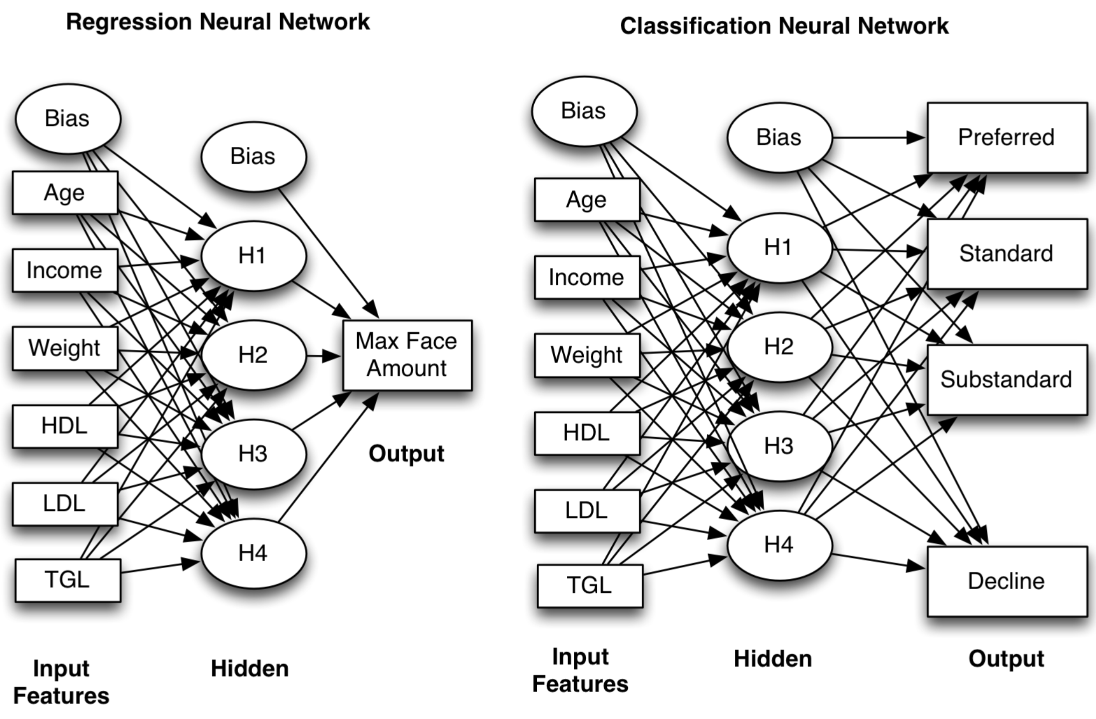


Figure 1. Regression and classification network (original features)

The left neural network performs a regression and uses the six original input features to set the maximum policy face amount to issue an applicant. The right neural network executes a classification and utilizes the same six input features to place the insured into an underwriting class. The weights (shown as arrows) establish the final output. A backpropagation algorithm fixes the weights through many sets of inputs that all have a known output. In this way, the neural network learns from existing data to predict future data. Furthermore, for simplicity, the above networks have a single hidden layer. Deep neural networks typically have more than two hidden layers between the input and output layers (Bengio, 2009). Every layer except the output layer can also receive a bias neuron that always outputs a consistent value (commonly 1.0). Bias neurons enhance the neural network's learning ability (B. Cheng & Titterington, 1994).

Output neurons provide the neural network's final outcome. Before the output neurons can be determined, the values of previous neurons must be calculated. The following equation can determine the value of each neuron:

$$\boldsymbol{f}(\boldsymbol{x},\boldsymbol{w},\boldsymbol{b}) = \boldsymbol{\phi}\left(\sum\nolimits_{\boldsymbol{i}}(\boldsymbol{w}_{\boldsymbol{i}}\boldsymbol{x}_{\boldsymbol{i}}) + \boldsymbol{b}\right) \tag{1}$$

The function *phi* (Φ) represents the transfer function, and it is typically either a rectified linear unit (ReLU) or one of the sigmoidal functions. The vectors *w* and *x* represent the weights and input; the variable *b* represents the bias weight. Calculating the weighted sum of the input vector (*x*) is the same as taking the dot product of the two vectors. This is why neural networks are often considered to be part of a larger class of machine learning algorithms that are dot product based.

For input neurons the vector *x* comes directly from the data set. Now that the input neuron values are known, the first hidden layer can be calculated, using the input neurons as *x*. Each subsequent layer is calculated with the previous layer's output as *x*. Ultimately, the output neurons are determined, and the process is complete.

Feature engineering adds calculated features to the input vector (Guyon, Gunn, Nikravesh, & Zadeh, 2008). It is possible to use feature engineering for both classification and regression neural networks. Engineered features are essentially calculated fields that are dependent on the other fields. Calculated fields are common in business applications and can help human users understand the interaction of several fields in the original data set. For example, insurance underwriters benefit from combining height and weight to calculate body mass index (BMI). Likewise, insurance underwriters often use a ratio of the HDL, TGL and LDL cholesterol levels. These calculations allow a single number to represent an aspect of the health of the applicant. These calculations might also be useful to the neural network. If the BMI and HDL/LDL ratios were engineered as features, the classification network would look like Figure 2.

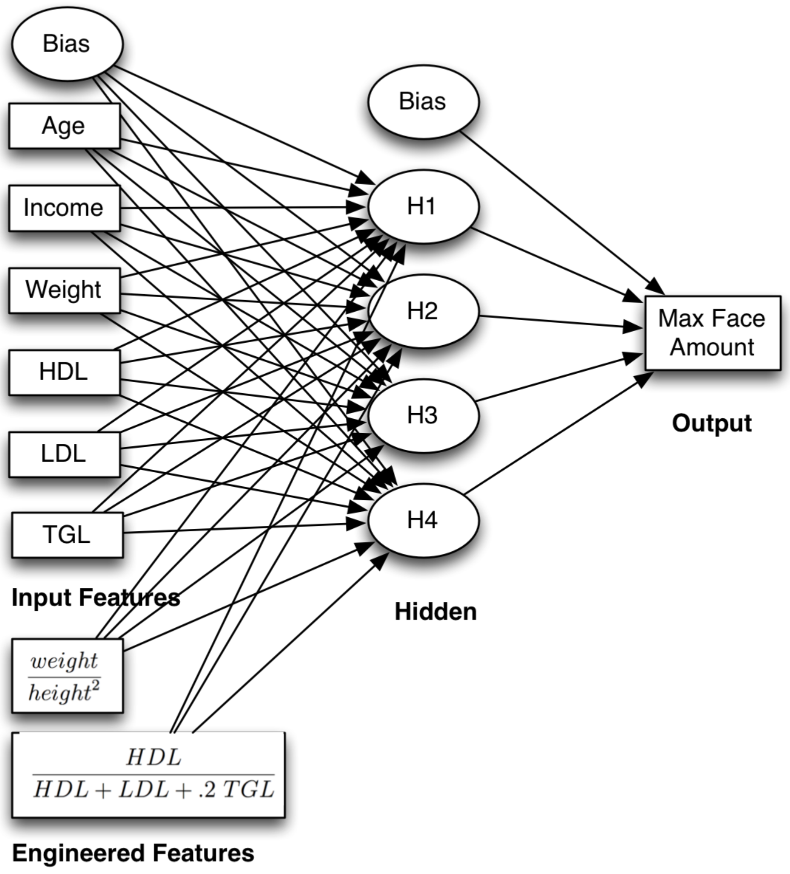


Figure 2. Neural network engineered features

In Figure 2, the BMI and HDL/LDL ratio values are appended to the feature vector along with the original input features. This calculation produces an augmented feature vector that is provided to the neural network. These additional features might help the neural network to calculate the maximum face amount of a life insurance policy.

Similarly, these two features could also augment the feature vector of a classification neural network. BMI and the HDL/LDL ratio are typical of the types of features that might be engineered for a neural network. Such features are often ratios, summations, and powers of other features. Adding BMI and the HDL/LDL ratio is not complicated because these are well-known calculations. Similar calculations might also benefit other data sets. Feature engineering often involves combining original features with ratios, summations, differences, and power functions. BMI is a type of engineered feature that involves multiple original features and is derived manually using intuition about the data set.

Problem Statement

There is currently no automated means of engineering features specifically designed for deep neural network that are a combination of multiple named features from the original vector. Previous automatic feature engineering work focused primarily upon the transformation of a single feature or upon models other than deep learning (Box & Cox, 1964; Breiman & Friedman, 1985; Freeman & Tukey, 1950). Although model-agnostic genetic programming-based feature extraction algorithms have been proposed (Guo, Jack, & Nandi, 2005; Neshatian, 2010), they do not tailor engineered features to deep neural networks. Feature engineering research for deep learning has primarily dealt with high-dimensional image and audio data (Blei, Ng, & Jordan, 2003; M. Brown & Lowe, 2003; Coates, Lee, & Ng, 2011; Coates & Ng, 2012; Le, 2013; Lowe, 1999; Scott & Matwin, 1999).

Machine learning performance depends on the representation of the feature vector. Feature engineering is an important but labor-intensive component of machine learning applications (Bengio, 2013). As a result, much of the actual effort in deploying machine learning algorithms goes into the design of preprocessing pipelines and data transformations (Bengio, 2013).

Deep neural networks (Hinton, Osindero, & Teh, 2006) can benefit from feature engineering. Most research into feature engineering in the deep learning field has been in the areas of image and speech recognition (Bengio, 2013). Such techniques are successful in the high-dimension space of image processing and often amount to dimensionality reduction techniques (Hinton & Salakhutdinov, 2006) such as Principal Component Analysis (PCA) (Timmerman, 2003) and auto-encoders (Olshausen & Field, 1996).

Dissertation Goal

The goal of this dissertation is to use genetic programming to analyze a data set of named features and to automatically create engineered features that will produce a more accurate deep neural network. These engineered features will consist of mathematical transformations of one or more of the existing features from the data set. Feature engineering will only improve accuracy when the engineered feature exposes an interaction that the neural network could not determine from the data set (W. Cheng, Kasneci, Graepel, Stern, & Herbrich, 2011). Consequently, feature engineering will not benefit all data sets. To mitigate this problem, it is important to evaluate several real world data sets, as well as synthetic data sets designed specifically to benefit from feature engineering.

The proposed research focuses on data sets consisting of named features, as opposed to data sets that contain large amounts of pixels or audio sampling data. Fraud monitoring, sales forecasting, and intrusion detection are predictive modeling applications where the input is made up of these named values. Real-world data sets that provide a basis for predictive modeling can consist of a handful or of several hundred such named features. Therefore, the proposed algorithm would be ineffective for the high-dimension tasks of computer vision and hearing applications. Such data sets are outside of the scope of this dissertation research.

Relevance and Significance

Since the introduction of multiple linear regression, statisticians have been employing creative means to transform input to enhance model accuracy (Anscombe & Tukey, 1963; Stigler, 1986). These transformations usually applied a single mathematical expression to an individual feature. For example, one feature might apply a logarithm; another feature might be raised to the third power. Transformations that apply an expression to a single original feature can significantly increase the accuracy of certain model types (Kuhn & Johnson, 2013).

Researchers have focused much attention on automated derivation of single-feature transformations. Freeman and Tukey (1950) reported on a number of useful transformations for linear regression. Box and Cox (1964) conducted the seminal work on automatic feature transformation and invented a stochastic ad hoc algorithm that recommends transformations that might improve the results of linear regression. Their work became known as the Box Cox transformation. Although the work by Box and Cox was capable of obtaining favorable transformations for linear regression, it often did not converge to the best one for an individual feature because of its stochastic nature. Numerous other transformations were created that were based on similar stochastic sampling techniques (Anscombe & Tukey, 1963; Mosteller & Tukey, 1977; Tukey, Laurner, & Siegel, 1982). Each of these algorithms focused on transformations of single features for the linear regression model.

Breiman and Friedman (1985) took a considerable step forward by introducing the alternating conditional expectation (ACE) algorithm that guaranteed optimal transformations for linear regression. Despite the fact that all of the aforementioned algorithms were designed for linear regression, they can also assist other model types. Because ACE was designed for linear regression, it cannot guarantee optimal transformations for other model types (Ziehe, Kawanabe, Harmeling, & Müller, 2001). Additionally, ACE transforms the entire feature vector by using separate transformations for each input feature, as well as the output. Engineered features do not need to come from the transformation of a single feature from the original vector; several original input features can be transformed collectively to produce favorable results (H.-F. Yu et al., 2011).

Feature engineering played an important role in several winning Kaggle and ACM’s KDD Cup submissions. H.-F. Yu et al. (2011) reported on the successful application of feature engineering to the KDD Cup 2010 competition. Ildefons and Sugiyama (2013) won the Kaggle Algorithmic Trading Challenge with an ensemble of models and feature engineering. A manual process created the features engineered for these competitions.

Barriers and Issues

It is a difficult and challenging problem to create an algorithm that automatically engineers features made up of one or more original named features and designed for a deep neural network. Most automated feature engineering algorithms are built for linear regression models. Furthermore, most of these automated methods typically engineer features that are based upon only a single feature from the original vector.

A linear regression model is a multi-term, first-degree expression. Because of its simplicity, several mathematical techniques can verify and assure optimal transformations. Deep neural networks have a much more complex structure than linear regression. Therefore, engineering features for a deep neural network presents different challenges than those for a linear regression.

This dissertation will seek an algorithm that combines multiple original features. As the number of original features considered increases, so does the search space of the algorithm. This expanded search space will face the curse of dimensionality (Bellman, 1957) and require novel solutions to limit and prioritize the search space. However, ignoring the curse of dimensionality in favor of a simple theoretical approach to automatic feature engineering can generate every possible expression based on the original feature set. Obviously, this is an infinite task. Even with a small number of features and one expression type, the search space is still huge. Therefore, it is useful to consider searching every combination of an engineered rational difference as demonstrated by the following expression:

$$\boldsymbol{\frac{a-b}{c-d}} \tag{2}$$

If the data set were modestly large and contained 100 features in the original data set, the complete search space would be over 47 million permutations:

$$\mathbf{0.5}\,\frac{{}_{\mathbf{100}}\mathrm{P}_{\mathbf{4}}}{\mathbf{2}} = \mathbf{0.5}\,\frac{\mathbf{100}!}{(\mathbf{100}-\mathbf{4})!} = \mathbf{4.705} \times \mathbf{10}^{\mathbf{7}} \tag{3}$$

The above permutation is divided by two in order to account for the fact that the resulting expression is algebraically the same as the first if the order of both the numerator and denominators differences are flipped. This example is only one of the many types of features that the algorithm would need to check. These expressions will have to be prioritized so that the most likely successful candidates are explored first. It is possible to prioritize candidate features for evaluation by the analysis of the importance of the original feature.

Although it is necessary to reduce the search space, finding an efficient means of evaluating candidate feature transformations is also crucial. An obvious test of a feature transformation would be to fit a neural network with the candidate feature and then fit a neural network without it. However, neural networks start from random weights, so it is advisable to fit several times and calculate the lowest error across those runs to mitigate the effects of a bad set of starting weights. It might also be advantageous to try several neural network hidden weight architectures to ensure that a candidate feature was not simply reacting badly to a poorly architected neural network. Obviously, it is not possible to provide such an exhaustive evaluation for each candidate feature. The evaluation code must be quick and able to run in parallel. Additionally, the neural networks must be structured so that they give the best assessment possible for each candidate-engineered feature. All of these problems must be overcome to give the best possible ranking of the candidate-engineered features.

The fact that not all data sets benefit from feature engineering creates another complication. As a result, it will be necessary to use a variety of data sets from the UCI Machine Learning Repository (Newman & Merz, 1998). It will also be essential to generate several synthetic data sets that are designed to benefit from certain types of engineered features. Considerable experimentation is required to try all selected real and synthetic data sets against the proposed solution.

Definitions of Terms

Alternating Condition Expectation (ACE): A form of automatic feature transformation that is applied to both the input and output of a linear regression model.

Auto-Encoder: A neural network that can learn efficient encodings for data to perform dimension reduction. An auto-encoder will always have the same number of input and output neurons.

Bias Neuron: A neuron that is added to the input and hidden layers that always applies a fixed weight. The bias neuron performs a similar function as the *y*-intercept in linear regression.

Box Cox Transformation: An automatic feature transformation for linear regression that can find helpful transformations for individual features.

Classification: A neural network, or other model, that is trained to classify its input into a predefined number of classes. A classification neural network will typically use a softmax function on its output and have a number of output neurons equal to the total number of classes.

Constant Node: A terminal node in a genetic program that holds a constant value.

Constant Pool: A fixed-length pool of numbers that can be assigned to constant nodes in a genetic program.

Cross Entropy Loss Function: A neural network loss function that often provides superior training results for classification when compared to the quadratic loss function. For regression problems, root mean square error (RMSE) should be considered.

Crossover: An evolutionary operator, inspired by the biological concept of sexual reproduction in which two genomes combine traits to produce offspring. Crossover performs the exploitation function of an evolutionary algorithm by creating new genomes based on fit parents.

Data Set: For supervised training, a data set is a collection of input values (*x*) and the expected output values (*y*). The presented research only deals with supervised training. The overall data set is usually divided into training and validation data sets.

Deep Learning: A collection of training techniques and neural network architecture innovations that make it possible to effectively train neural networks with three or more layers.

Deep Neural Network: A neural network with three or more layers.

Directed Acyclic Graph (DAG): A graph where all connections contain directions and there are no loops. The genetic programs found in this research are represented as DAGs.

Dot Product Based Model: A model that uses dot products, or weighted sums, as a primary component of their calculation. Neural networks, linear regressions, and support vector machines for regression are all examples of dot product based models.

Dynamic Constant Node: A special type of constant node found in Encog genetic programs that can change its value as genetic programming training progresses.

Contrasted with the constant pool found in many genetic programming frameworks.

Elitism: When a certain number of the top genomes of a population are chosen to always survive into the next generation.

Engineered Feature: A feature that is added to the feature vector of a model, such as a neural network, that is a mathematical transformation of one or more features from the original feature vector.

Evolutionary Algorithm: An optimization algorithm that evolves better-suited individuals from a population by applying mutation and crossover. The algorithm must balance between exploring new solutions and exploiting existing solutions to make them better.

Expression: A mathematical representation that involves operators, constants, and variables. Expressions are the genomes that the algorithm manipulates in this dissertation.

Exploitation: The process in a search algorithm where the evaluation is constrained to regions close to the best solution discovered so far.

Exploration: The process in a search algorithm where the search is broadened to new regions farther away from the best solution discovered so far.

Feature Engineering: The process of creating new features by applying mathematical transformations to one or more of the original features.

Feature Importance: A measurement of the importance of a single feature to the neural network, relative to the other features.

Feature Selection: The process of choosing the most important features to the neural network.

Feature Vector: The complete set of inputs to a neural network. The feature vector must be the same size as the input layer.

Feature: A single value from the feature vector of a neural network or other model. The features in a feature vector organize themselves around the neurons in the input layer of a neural network.

Feedforward Neural Network: A neural network that contains only forward connections between the layers.

Gated Recurrent Unit (GRU): A computationally efficient form of Long Short-Term Memory (LSTM) that combines several gates into one.

Genetic Programming: An evolutionary algorithm that develops programs to optimize their output for an objective function.

Genome: An individual in an evolutionary program.

Gradient: A partial derivative of the loss function of a neural network with respect to a single weight. The gradient is a key component in many neural network-training algorithms.

Hidden Layer: A neural network layer that occurs between the input and output layers. A neural network can contain zero or more hidden layers.

Input Layer: The first layer of neurons that receives the input to the neural network. Neural networks have a single input layer that must be the same length as the input vector to the neural network.

Interior Node: A node in a genetic program that is not a terminal node or root node; it has both parents and children.

Kaggle: A website where individuals compete to achieve the best model for posted data sets. Kaggle is considered as a source of benchmarking data sets in this research.

KDD Cup: An annual competition, hosted by the Association for Computing Machinery (ACM) where individuals compete for the best model fit on a provided data set.

Latent Dirichlet Allocation (LDA): A natural language processing generative statistical model.

Layer: A collection of related neurons in a neural network. A neural network typically has an input, output, and zero or more hidden layers.

Learning Rate: A numeric constant that controls how quickly a model, such as a neural network, learns. For backpropagation the learning rate is multiplied by the gradient to determine the value to change the weight.

Linear Discriminant Analysis (LDA): Generalization of the classification algorithm originally proposed by Fisher for the iris data set.

Linear Regression: A simple model that computes its output as the weighted sum of the input plus an intercept value.

Log Loss Error: A common error metric for classification of neural networks.

Long Short-Term Memory (LSTM): A type of recurrent neural network that uses a series of gates to learn patterns spanning much larger sequences than those that regular simple recurrent networks are capable of learning.

Loss Function: A function that measures the degree to which the actual input from a neural network ($\hat{y}$) is different than the expected output (y).

Momentum: A numeric constant that attempts to prevent a neural network from falling into a local minimum. This value is multiplied by the previous iteration's weight change to determine a value to add to the current iteration's weight change.

Mutation: An evolutionary operator, inspired by the biological concept of asexual reproduction, in which a single genome produces offspring with a slightly altered set of traits than the single parent. Because mutation introduces new stochastic information, it fulfills the exploration component of an evolutionary algorithm.

Named Feature: A description used in this dissertation for a feature that represents a specific value, such as a measurement or a characteristic of the data. An image is a high-dimension input that would contain pixels, as opposed to named features.

Natural Language Processing: Artificial intelligence algorithms that are designed to understand human language.

Nesterov Momentum: A more advanced form of classic momentum that attempts to mitigate the effects of over correcting.

Neural Network: A model inspired by the human brain that is composed of an input layer, zero or more hidden layers, and an output layer.

Objective Function: A function that evaluates a genetic program (genome) and produces a score. The evolutionary algorithm will try to create genomes that either maximize or minimize this score. Most evolutionary algorithms can be configured to either maximize or minimize.

One-Versus-Rest: A technique that allows multiple binary classification models to perform multiclass classification by training one classification model per class to classify between that class and the rest of the classes.

Output Layer: The final layer of a neural network. This layer produces the output. A regression neural network will typically have a single output neuron. A binary classification neural network will also have a single output neuron. A classification neural network with three or more classes will have an output neuron for each class.

Preprocessing: An algorithm that prepares data for a neural network or other model. The feature engineering explored in this paper would function as a part of data preprocessing for a neural network.

Principal Component Analysis (PCA): A form of dimension reduction that can shrink the size of an input vector to a smaller encoding with minimal loss of accuracy to the neural network.

Quadratic Loss Function: A simple neural network loss function that uses the difference between the expected output and actual output of a neural network. The quadratic loss function should be the first choice for regression neural networks; however, the cross entropy loss function should be the choice for classification neural networks.

Recurrent Neural Network: A neural network that contains backwards connections from layers to previous layers.

Regression: A neural network or other model that is trained to produce a continuous value as its output. A regression neural network will use a linear transfer function on its output and have a single output neuron.

Root Mean Square Error: A neural network loss function that is typically found in regression problems.

Root Node: The node in a tree that is an ancestor of all other nodes. The root node for a single-node tree is also a terminal node.

Selection: An algorithm that chooses fit genomes for evolutionary operations such as crossover and mutation.

Sigmoidal: Something that is s-shaped.

Simple Recurrent Network (SRN): A network with only a single recurrent connection such as an Elman or Jordan network.

Softmax: An algorithm that ensures that all outputs of a neural network sum to 1.0, thereby allowing the output to be considered as probabilities.

Stochastic Gradient Descent (SGD): A variant of the backpropagation algorithm that uses a mini-batch that is randomly sampled each training iteration. SCG has proven itself to be one of the most effective training algorithms, and it is the neural network training method for the research proposed for this dissertation.

Symbolic Expression (S-Expression): Notation for nested list (tree-structured) data, invented for the Lisp programming language, which uses it for source code as well as data.

Synthetic Data set: A data set that was generated to test a specific characteristic of an algorithm.

Terminal Node: A node in a tree that has no children. Terminal nodes are also referred to as leaf nodes.

Training Data set: The data on which the model was actually trained. Usually, validation data are also kept so that the model can be evaluated on different data than it was trained with.

Transfer Function: Applied to the weighted summations performed by the layers of a neural network. All layers of a neural network have transfer functions except the input layer. Transfer functions are sometimes referred to as activation functions.

Tree-Based Genetic Program: A genetic program that is represented as a tree of nodes. The research proposed by this dissertation uses tree-based genetic programs.

Turing Complete: A system of data-manipulation rules that simulates any single-taped Turing machine. Also referred to as computationally universal.

Universal Approximation Theorem: A theorem that states that a feedforward network with a single hidden layer containing a finite number of neurons can approximate continuous functions.

Validation Data set: The portion of the data set that validates model predictions on data that are outside of the training data set. This data set is sometimes referred to as out-of-sample data.

Xavier Weight Initialization: A neural network weight initialization algorithm that produces relatively quick convergence for backpropagation and limited variance of required iteration counts for repeated training of a neural network.

YAML Ain't [sic] Markup Language (YAML): A common configuration file format that communicates operating parameters to the algorithm proposed by this research.

List of Acronyms

ACE: Alternating Condition Expectation

ANN: Artificial Neural Network

CGP: Cartesian Genetic Programming

DAG: Directed Acyclic Graph

DBNN: Deep Belief Neural Network

DNN: Deep Neural Network

GP: Genetic Programming

GRU: Gated Recurrent Unit

IDS: Intrusion Detection System

KDD: Knowledge Discovery in Databases

LDA: Latent Dirichlet Allocation or Linear Discriminant Analysis

LSTM: Long Short-Term Memory

NEAT: NeuroEvolution of Augmenting Topologies

ReLU: Rectified Linear Unit

RDBMS: Relational Database Management System

SGD: Stochastic Gradient Descent

SRN: Simple Recurrent Neural Network

SIFT: Scale-Invariant Feature Transform

TD-IDF: Term Frequency–Inverse Document Frequency

T-SNE: t-Distributed Stochastic Neighbor Embedding

XOR: Exclusive Or

YAML: YAML Ain't [sic] Markup Language

Summary

This chapter introduced proposed research into automated feature engineering. This algorithm will leverage the ability of genetic programming to generate expressions that might become useful features for neural networks. The features engineered by this algorithm will consist of expressions that utilize one or more features from the data set's original feature vector. The resulting engineered features should increase the accuracy of the neural network for some data sets.

The primary challenge of this research is to limit the infinite search space of expressions that combine the features of the data set. This goal is accomplished by defining the types of expressions that most benefit a neural network, determining the importance of a feature, and prioritizing the expressions to be searched by the algorithm. It would also be beneficial if the algorithm could run in parallel to utilize multiple cores on the host computer system.

Additionally, the algorithm needs an efficient objective function to evaluate candidate-engineered features against each other. Such a function will likely be based on fitting a neural network model with candidate-engineered features. This objective function must be designed efficiently so that it can determine in minimal time how effective one candidate engineered feature is compared to another. Genetic programming uses this type of objective function to decide the best genomes (candidate-engineered features) to form the next generation of genomes.

The remainder of this dissertation proposal is organized as follows: Chapter 2 provides a review of the literature that directly influenced this research. Chapter 3 presents the methodology applied in order to implement an algorithm for automated feature engineering.

Chapter 2

Literature Review

The research proposed for this dissertation focuses primarily upon feature engineering and how to apply it to deep neural networks. Because of its ability to manipulate expressions, genetic programming will enable the dissertation algorithm to recommend engineered features. The following areas of literature are important to the proposed research:

- Feature engineering
- Neural networks
- Deep learning
- Genetic programming

There is considerable research community interest in all of these areas. The following sections review current literature in these areas as it pertains to the proposed research:

Feature Engineering

The input vector of a predictive model can be augmented or transformed to enhance predictive performance. In literature, this process is often referred to as feature engineering, feature modification, or feature extraction. Automated variants of these processes are sometimes referred to as automated feature engineering or feature learning. For consistency, this dissertation will use the terminology of feature engineering to refer to this augmentation. It will use automated feature engineering to refer to an algorithm that automates this feature engineering.

These techniques grew out of the need to transform model input into forms conducive for linear regression (Freeman & Tukey, 1950). This transformation can be helpful to lower residual error on a linear regression. Box and Cox (1964) showed a method for determining which of several power functions might be a beneficial feature transformation for linear regression input. The algorithm in their paper became known as the Box-Cox transformation. Power transformations simply apply exponents to the input features of a machine learning model. Other mathematical functions may also perform transformation. Logarithms are a popular choice. Linear regression is not the only machine learning model that benefits from feature engineering transformations. These simple transformations modify the individual features independently of each other.

The Box and Cox (1964) relied upon a stochastic sampling of the data and does not necessarily guarantee an optimal set of transformations. Breiman and Friedman (1985) introduced the alternating conditional expectation (ACE) algorithm that could ensure optimal transformations for linear regression. The ACE algorithm finds a set of optimal transformations for each of the predictor features, as well as the output for linear regression. Although the resulting transformations were originally intended for linear regression, they work for other model types as well (B. Cheng & Titterington, 1994).

Splines are a common means of feature transformation for most machine learning model types. By fitting a spline to individual features, it is possible to smooth the data and reduce overfitting. The number and position of knots inside the spline is a hyper-parameter that must be determined for this transformation technique. Splines have the capability of taking on close approximations of the shape of other functions. Brosse, Lek, and Dauba (1999) used splines to transform data for a neural network to predict the distribution of fish populations.

Machine vision has been a popular application of feature engineering. A relatively early form of feature engineering for computer vision was the Scale-Invariant Feature Transform (SIFT) (Lowe, 1999). This transformation attempts the recognition of images at multiple scales, which is a common problem in computer vision. A machine learning model that learns to recognize digits might not perceive these same digits if their size is doubled. SIFT preprocesses the data and provides them in a form where images at multiple scales produce features that are similar. These types of features can be generalized for many problems and applications in machine vision, including object recognition and panorama stitching (M. Brown & Lowe, 2003).

Text classification is another popular application of machine learning algorithms. Scott and Matwin (1999) utilized feature engineering to enhance the performance of rules learning for text classification. These transformations allow structure and frequency of the text to be generalized to a few features. Representing textual data to a machine learning model produces a considerable number of dimensions. Text classification commonly uses feature engineering to reduce these dimensions.

Another application of feature engineering to text classification is the latent dirichlet allocation (LDA) engineered feature. This method transforms a corpus of documents into document-topic mappings (Blei et al., 2003). LDA has subsequently been applied to spam filtering, among several document classification tasks (Bíró, Szabó, & Benczúr, 2008) and article recommendation (C. Wang & Blei, 2011).

Many data are stored in relational database management systems (RDBMS) and consist of a number of different tables that form links, or relations, between them. The relationships between these tables can be of various cardinalities, leading to relationships including one-to-one, one-to-many, or many-to-many. Machine learning models typically require a fixed-length feature vector. Mapping the RDBMS linked data into a feature vector that is suitable for a machine learning model can be difficult. Automated mapping of RDBMS data is an active area of research. Bizer, Heath, and Berners-Lee (2009) created a system where the data are structured in a way that they can be accessed with semantic queries.

Feature engineering has proven to be valuable in the Kaggle and KDD Cup data science competitions. In fact, one team utilized feature engineering and an ensemble of machine learning models to win the KDD Cup 2010 competition (H.-F. Yu et al., 2011). Histograms of oriented gradients were other features presented in this competition. W. Cheng et al. (2011) developed an automated feature generation algorithm for data organized with domain-specific knowledge. These technologies have found many applications. For example, Ildefons and Sugiyama (2013) were able to win the Kaggle Algorithmic Trading Challenge with an ensemble of models and feature engineering. The features engineered for these competitions were created manually or with knowledge about the specific competition problem. Such knowledge is referred to as domain-specific knowledge and requires human intuition that cannot currently be replicated by a machine.

Feature engineering has also advanced natural language processing (NLP). An example of an engineered feature for NLP is the term frequency inverse document frequency (TF-IDF). This engineered feature is essentially the ratio of the frequency of a word in a document compared to its occurrence in the corpus of documents (Rajaraman & Ullman, 2011). TF-IDF has proven popular for text mining, text classification, and NLP.

Researchers have examined the ability of machine learning algorithms to perform automated feature learning. These algorithms are often unsupervised in that they examine the data without regard to an expected outcome. Coates et al. (2011) implemented an unsupervised single-layer neural network for feature engineering. Principal component analysis (PCA) (Timmerman, 2003) and t-distributed stochastic neighbor embedding (T-SNE) (Van der Maaten & Hinton, 2008) are dimension-reduction algorithms that have also proven to be successful for automated feature engineering. Other unsupervised machine learning algorithms have also been applied to feature engineering. Coates and Ng (2012) utilized k-means clustering for feature engineering.

Deep neural networks have many different layers to learn complex interactions in the data. Despite this advanced learning capability, deep learning also benefits from feature engineering. Bengio (2013) demonstrated that feature engineering is useful for speech recognition, computer vision, classification, and signal processing. Le (2013) engineered high-level features using unsupervised techniques to construct a deep neural network for signal processing.

Lloyd, Duvenaud, Grosse, Tenenbaum, and Ghahramani (2014) employed feature engineering to create the *Automatic Statistician* project. This system spontaneously models regression problems and produces readable reports. This system can determine the types of transformations that might benefit individual features.

Kanter and Veeramachaneni (2015) invented a technique called deep feature synthesis that automatically transforms relational database tables into the feature vector needed by the typical machine learning model. The feature vector input to a neural network cannot directly encode the one-to-many and many-to-many relationships that are common in RDBMSs. The deep feature synthesis algorithm employs SQL-like transformations, such as MIN, MAX, and COUNT to summarize and encode these relationships relationships into a feature vector. The authors of deep feature synthesis reported on their algorithm's ability to outperform some competitors in three data science competitions.

Researchers have implemented automated feature engineering for specific domains. Davis and Foo (2016) applied automated feature detection to modeling tasks involving HTTP traffic and tunnels. Cuayáhuitl (2016) created SimpleDS, a system for text document processing that avoids manual feature engineering by using a deep reinforcement learning system. Manual feature engineering remains popular, and researchers have explored its value in various contexts. Bahnsen, Aouada, Stojanovic, and Ottersten (2016) provide guidelines and examples of feature engineering for credit card fraud detection. This is an example of domain-specific knowledge. Zhang, Huan, and Jiang (2016) investigated feature engineering for phishing detection. Although these systems are effective in their specific domains, they do not address the problem statement proposed by this research of creating a generic automated feature engineering system.

Neural Networks

Neural networks are a biologically-inspired class of algorithms that McCulloch and Pitts (1943) introduced as networks composed of MP-Units. Although neural networks contain connections and neurons, they do not attempt to emulate every aspect of real world neurons. Modern neural networks are more of a mathematical model than a biological simulator. The seminal neural network algorithm of McCulloch and Pitts specifies the calculation of a single neuron, called an MP-Unit, as the weighted sum of the neuron's inputs. This weighted sum is a mathematical dot product. Nearly all neural networks created since their introduction in 1943 are based upon feeding dot product calculations to transfer functions over layers of neurons. Deep neural networks simply have more layers of neurons (MP-Units).

Initially, the weights of neural networks were handcrafted to create networks capable of solving simple problems. The research community has shown great interest in automating neural network weight selection to achieve a particular objective. Hebb (1949) defined a process to describe how the connection strengths between biological neurons change as learning occurs. When the organism performs actions, connections increase between the neurons necessary for that action. This process became Hebb's rule, and it is often informally stated as, "neurons that fire together wire together."

Rosenblatt (1962) introduced the perceptron that became the seminal neural network that contained input and output layers. The perceptron is a two-layer neural network with an input layer that contains weighted forward-only connections to an output layer. The transfer function defined for the classic perceptron is a simple function that performs a threshold—it returns the value 1 if the neuron's weighted inputs reach a value above a specified threshold; otherwise, it returns the value 0. Minsky and Papert (1969) described severe limitations in the perceptron in their monograph. They demonstrated that perceptrons were incapable of learning the Exclusive Or (XOR) operator, a non-linearly separable problem.

Research continued for the automatic derivation of the weights of a neural network, beginning with the work of Werbos (1974) when he mentioned that gradient descent could be used for the training of neural networks in his Ph.D. thesis. Previously, researchers used gradient descent to find minimums of functions. Rumelhart, Hinton, and Williams (1985) were the first to apply gradient descent to neural network training. Their algorithm is called the backward propagation of errors, or backpropagation. The gradient of each weight is calculated and determines a change that should occur in the weight for the current training iteration. The gradient of each weight is the partial derivative of the loss function for that weight with all other weights held constant. Therefore, backpropagation applies gradient descent to neural network training.

Backpropagation was initially ineffective at training neural networks with significantly more than two hidden layers (Bengio, 2009). Furthermore, it was not known if neural networks actually benefited from many layers. Gybenko (1989) formulated the universal approximation theorem and proved that a single hidden-layer neural network could approximate any function. Hornik (1991) continued this research by showing that the multilayer feedforward architecture – and not the specific choice of the transfer function – gave neural networks the potential of being universal approximators. The universal approximation theorem implies that additional hidden layers are unnecessary because a single hidden-layer neural network can theoretically learn any problem. Although feedforward neural networks are universal approximators, they are not Turing complete (Graves, Wayne, & Danihelka, 2014; Turing, 1936) without extension. In other words, a feedforward neural network can emulate any function, but the neural network cannot replicate the operation of any computer program.

Just as for other statistical models, it is often necessary to explain why a neural network produced the output that it did. Garson (1991) created an algorithm that could rank the importance of the input neurons in order to reveal the behavior of neural networks. Goh (1995) proposed a similar feature-ranking algorithm that analyzed the weights of a neural network. In fact, many models have feature-ranking algorithms that analyze that individual model. These ranking algorithms are called model dependent because they work only for a single type of model.

Breiman (2001) introduced permutation feature importance in his seminal paper on random forests. Although he presented this algorithm in conjunction with random forests, it is model-independent and appropriate for any supervised learning model. Consequently, permutation feature importance and neural network weight analysis will play a role in this dissertation. The ranking of features that the proposed research will yield can become a score that functions as the genetic programming objective function.

By definition, a feedforward neural network contains only forward connections. Thus, the input layer connects only to the first hidden layer, the first hidden layer connects only to the second hidden layer, and the final hidden layer connects only to the output layer. However, recurrent neural networks allow connections to previous layers. Elman (1990) and Jordan (1997) began the research of recurrent neural networks and introduced their simple recurrent networks (SRNs) as the Elman and Jordan neural networks. Figure 3 shows an Elman neural network while Figure 4 shows a Jordan neural network.

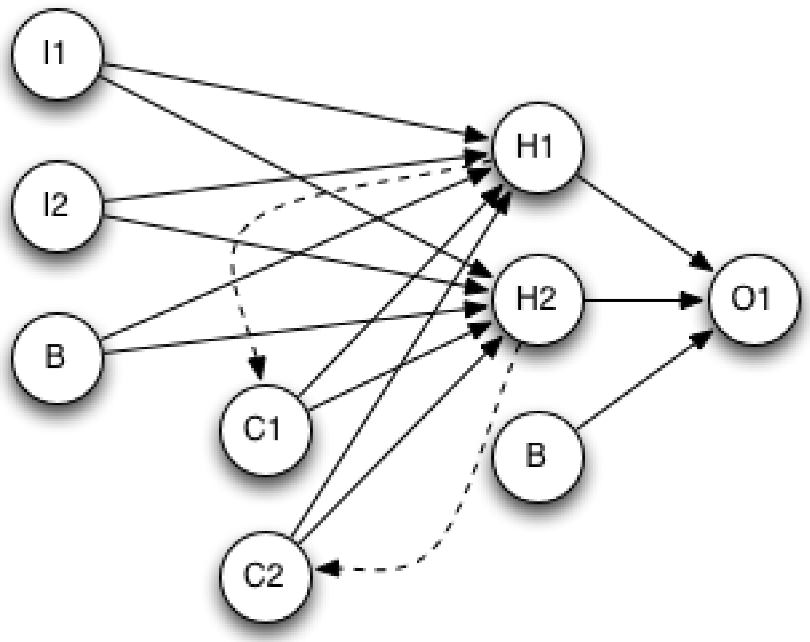


Figure 3. Elman neural network

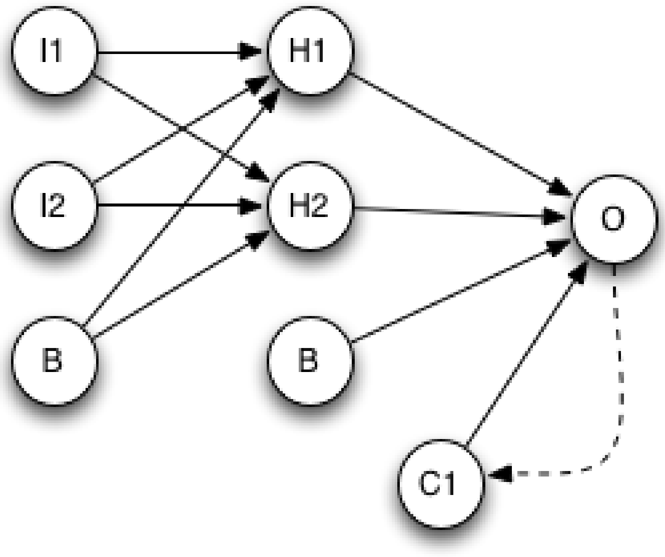


Figure 4. Jordan neural network

Both the Elman and Jordan networks contain a context layer (C) that will initially output 0. However, the context layer always remembers the input from the previous time that the network was calculated. Subsequent calculations of the neural network will cause the context layer to always output the input received by the layer in the previous iteration. The data received are stored and then output at the next calculation of the neural network. The context layer in a SRN can function somewhat like a time-loop in that the output is always the input from the previous iteration.

A technique known as backpropagation through time can usually train recurrent neural networks (Mozer, 1989; Robinson & Fallside, 1987; Werbos, 1988). This technique is similar to backpropagation except that it unfolds the recurrent layers to make the recurrent neural network appear as one large feedforward network. Backpropagation through time has a configuration parameter that specifies the number of time slices that the program can unfold into the network. A number of virtual layers equal to that configuration parameter can create the virtual network. The same backpropagation algorithm that was used for feedforward networks trains the virtual network.

Feedforward neural networks will always produce the same output for a given feature vector. However, recurrent neural networks will maintain state from previous computations. This state will affect the neural network output; therefore, the order that feature vectors are presented to the neural network will affect the output. This capability makes recurrent neural networks applicable to time series prediction. For recurrent neural networks, a series of events, represented as individual feature vectors, now produces an output. This result differs from a single feature vector in regular feedforward neural networks. While recurrent neural networks are particularly adept at handling time-series, there are other alternatives, Balkin and Ord (2000) demonstrated encoding methods to use regular non-recurrent feedforward neural networks with time series prediction.

One issue with SRN networks, such as Elman and Jordan, is that the longer a time series becomes, the less relevant a context layer. To overcome this problem, Hochreiter and Schmidhuber (1997) introduced the long short-term memory (LSTM) network, which is shown in *Figure 5*.

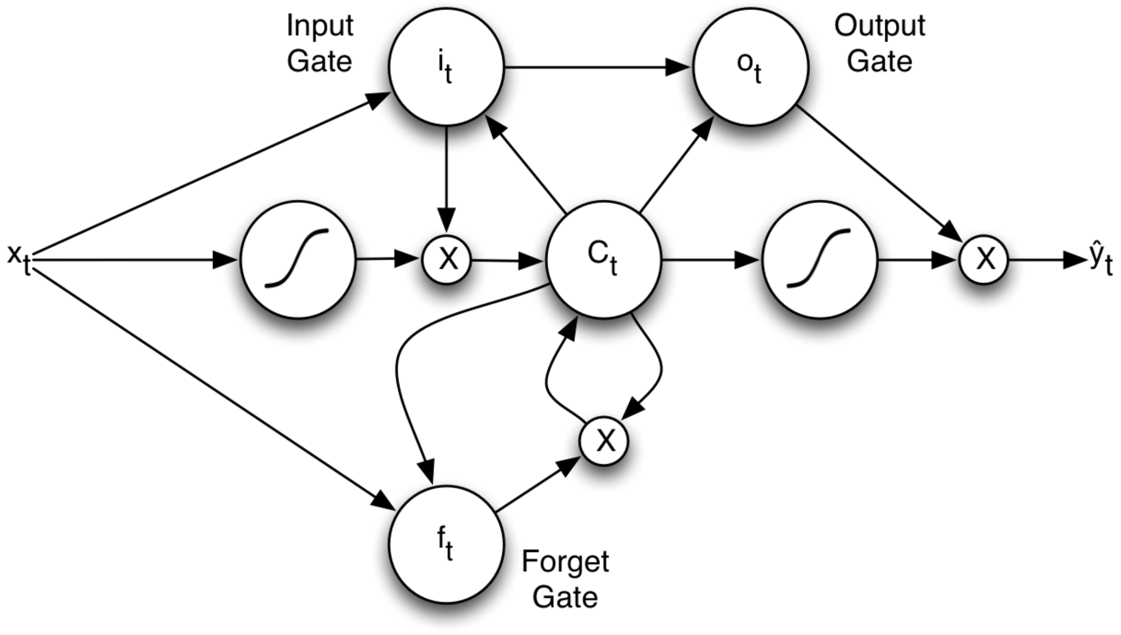


Figure 5. Long short-term memory (LSTM)

It is important to note that this figure shows only a single neuron of a LSTM network. These LSTM neurons can be placed inside of regular feedforward neural networks. Usually, LSTM neurons are placed as an entire layer of such neurons. On a conceptual level, the LSTM neuron functions similarly to the context neurons of the SRNs. The C-labeled node, near the center of the figure, represents the context memory of the node. However, unlike SRN's, the LSTM does not simply copy the previous neural network's computation to its internal state. The LSTM uses three gates to control when the input is accepted, when the internal state is forgotten, and when the internal state is output. These gates are activated when the input reaches a threshold specified by a trained internal weight. The backpropagation through time algorithm trains the gate threshold weights along with every other weight. Because it controls the internal state, input, and output, the LSTM is considerably more effective at recalling time series than a regular SRN.

The inability to learn large numbers of hidden layers was not the only barrier to widespread neural network adoption. One problem that remains for neural networks is the large number of hyper-parameters that a neural network contains. A neural network practitioner must decide the number of layers for the network as well as the quantity of neurons that each of the hidden layers will contain. Prior research in the field of neural networks reveals that researchers have long aspired to create an algorithm that automatically determines the optimal structure for neural networks. Stanley and Miikkulainen (2002) invented the NeuroEvolution of Augmenting Topologies (NEAT) neural network that utilizes a genetic algorithm to optimize the neural network structure. The genetic algorithm searches for the best neural network structure and weight values to minimize the loss function.

Although most neural network research has shifted towards deep learning, some research remains on classical neural network structures. Chea, Grenouillet, and Lek (2016) used a self-organizing map (SOM) to predict water quality. A SOM allowed M. Wang et al. (2016) to identify the mixtures of Chinese herbal medicines. Sobkowicz

(2016) implemented a NEAT neural network to perform sentiment analysis in the Polish language.

Deep Learning

While it is theoretically possible for a single-hidden layer neural network to learn any problem (Hornik, 1991), this outcome will not necessarily occur in practice. Additional hidden layers can allow the neural networks to learn hierarchies of features, thereby simplifying the search space. They can also find the optimal set of weights with less training. Unfortunately, no method to train these networks existed until the development of a series of innovations that introduced new transfer functions and training methods. McCulloch and Pitts (1943) introduced the seminal neural network calculation, shown in Equation 1. Many new technologies have built upon this core calculation.

Hinton et al. (2006) first implemented a deep neural network with favorable results. They created a learning algorithm that could train deep network variants like those that Fukushima (1980) introduced for belief networks. This discovery renewed interest in deep neural networks. Several additional technologies, such as stochastic gradient descent (SGD) (Bertsekas, 1999), rectified linear units (ReLU) (Glorot, Bordes, & Bengio, 2011), and Nesterov momentum (Sutskever, Martens, Dahl, & Hinton, 2013), have made training of deep neural networks more efficient. Taken together, these technologies are referred to as deep learning.

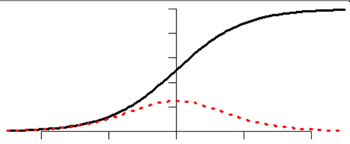
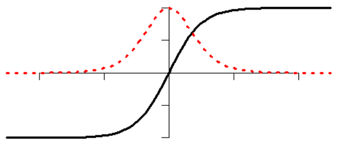
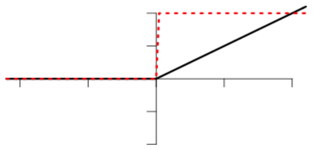
Neural networks with many layers will often experience a problem in which the gradients become 0 with certain transfer functions. Hochreiter (1991) was the first to describe this vanishing gradient problem in his Ph.D. thesis. Prior to deep learning, most neural networks used a simple quadratic error function on the output layer (Bishop, 1995). De Boer, Kroese, Mannor, and Rubinstein (2005) introduced the cross entropy error function, and it often achieves better results than the simple quadratic because it addresses the vanishing gradient problem by allowing errors to change weights even when neuron's gradient saturates (their derivatives are close to 0). It also provides a more granular means of error representation than the quadratic error function for classification neural networks. Therefore, the research presented in this dissertation will utilize the cross entropy error function for classification and root mean square error (RMSE) for regression.

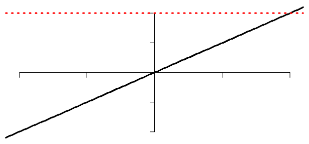
Neural networks must start with random weights (Bengio, 2009). These random weights are frequently sampled within a specific range, such as (-1,1). However, this simple range initialization can occasionally produce a set of weights that are difficult for backpropagation to train. As a result, researchers have shown interest for weight initialization algorithms that provide a good set of starting weights for backpropagation (Nguyen & Widrow, 1990). Glorot and Bengio (2010) introduced what has become one of the most popular methods called the Xavier weight initialization algorithm. Because of its ability to produce consistently performing weights suitable for backpropagation training, the research in this dissertation will use the Xavier weight initialization algorithm.

Backpropagation relies on the derivatives of the transfer functions to incrementally calculate, or propagate, error corrections from the output neurons back through the weights of a neural network. Prior to 2011, most neural network hidden layers used the hyperbolic tangent or the logistic transfer function, which are sigmoidal transfer functions. The derivative of both of these functions saturate to 0 as *x* approaches either positive or negative infinity, causing these transfer functions to exhibit the vanishing gradient problem. Glorot et al. (2011) introduced the rectified linear unit (ReLU) transfer function to address this problem.

The ReLU transfer function usually achieves better training results for deep neural networks than the sigmoidal transfer functions. According to current research (Bastien et al., 2012), the type of transfer function to use for deep neural networks is well defined for each layer type. For their hidden layers, deep neural networks employ the ReLU transfer function. For their output layer, most deep neural networks utilize a linear transfer function for regression, and a softmax transfer function for classification. No transfer function is needed for the input layer. Bastien et al. (2012) present research that follows this form and uses the ReLU transfer function for hidden layers and either linear or softmax for the output layer. Table 1 summarizes the logistic, hyperbolic tangent, ReLU, linear, and softmax transfer functions:

Table 1. Common neural network transfer functions

Name/ Range	Expression (Forward)	Derivative (Backward)	Graph (Derivatives in Red)
Logistic/ Sigmoid [0,1]	$\phi(x) = \frac{1}{1+e^{-x}}$	$\phi'(x) = \frac{e^{x}}{(1+e^{x})^2}$	
HTan [-1,1]	$\phi(x) = \text{htan}(x)$	$\phi'(x) = 1 - \phi^2(x)$	
ReLU [0,+∞)	$\phi(x) = \max(0, x)$	$\phi'(x) = \begin{cases} x > 0 & 1 \\ x \le 0 & 0 \end{cases}$	

Linear (-∞,+∞)	$\phi(x) = x$	$\phi'(x) = 1$	
Softmax (-∞,+∞)	$\phi(x)_j = \frac{e^{x_j}}{\sum_{k=1}^{\lVert x \rVert} e^{x_k}}$	NA	NA

This table indicates why ReLU often achieves better performance than a logistic or hyperbolic tangent. The graph column of the table shows both the transfer function as well as the derivative of that transfer function. The solid black line refers to the transfer function output, and the dotted red line is the derivative. Both sigmoid-shaped transfer functions have derivatives that quickly saturate to 0 as it approaches either positive or negative infinity. The ReLU, on the other hand, does not saturate as positive infinity is approached. Additionally, the much wider range of the ReLU lessens the need for the common practice of normalizing the inputs to values closer to the range of the sigmoidal transfer function in use.

Overfitting is a frequent problem for neural networks (Masters, 1993). A neural network is said to be overfit when it has been trained to the point that the network begins to learn the outliers in the data set. This neural network is learning to memorize, not generalize (Russell & Norvig, 1995). A class of algorithms designed to combat overfitting is called regularization algorithms. One of the most common forms of neural network regularization is to simply add a scaled summation of the weights of the neural network to the loss function. This calculation will cause the training algorithm to attempt to lower the weights of the neural network along with the output error. Two of the most common forms of this weight regularization are L1 and L2 (Ng, 2004).

L1 regularization, shown in the following equation, sums the weights of the neural network (*w*) and produces an error value (*E*) that is added to the loss function of the neural network.

$$\boldsymbol{E_1 = \frac{\lambda_1}{n} \sum_{w} |w|} \tag{4}$$

It is important to note that the *w* vector includes only actual weights and not bias-weights. The value λ is a scaling factor for the effect of the L1 regularization. If λ is too high, the objective of lowering the weights will overwhelm the one for achieving a lower error for the neural network training. This situation causes a failure of the neural network to converge to a low error. The value *n* represents the number of training set elements. L2 regularization is defined similarly to L1 and is provided by the following equation:

$$\boldsymbol{E_2 = \lambda_2 \sum_{w} w^2} \tag{5}$$

Both L1 and L2 regularization sum the weights without regard to their sign. This magnitude-oriented approach is accomplished by an absolute value for L1 and a square for L2. The weights are pushed towards 0 in both cases. However, L1 has a greater likelihood of pushing the weights entirely to 0 and effectively pruning the weighted connection (Ng, 2004). This pruning feature of L1 is especially interesting for the research proposed by this dissertation because it can function as a type of feature selection. L1 will indicate worthless engineered features by pruning them.

L1 and L2 are not the only forms of regularization. Srivastava, Hinton, Krizhevsky, Sutskever, and Salakhutdinov (2014) introduced dropout as a simple regularization technique for deep neural networks. Dropout is typically implemented as a single dropout layer, as demonstrated by Figure 6.

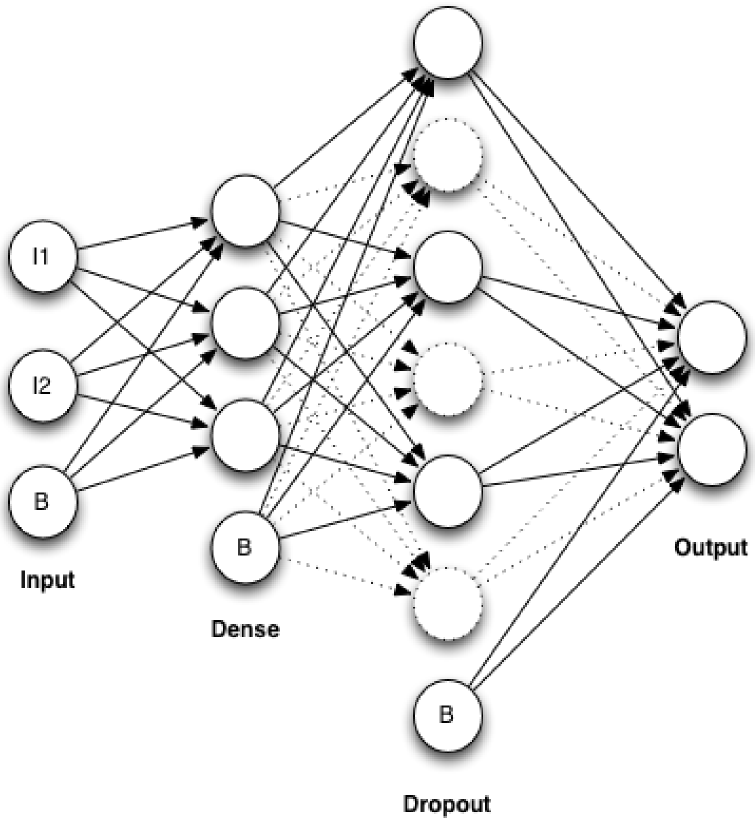


Figure 6. Dropout layer in a neural network

The dashed lines in Figure 6 represent the dropped neurons. In each training iteration, some neurons are removed from the dropout layer. However, neither the bias neurons nor the input and output neurons are ever removed. When a neuron is discarded, the training iteration occurs as if that neuron and all of its connections are not present. However, the drop is only temporary; the neuron and its connections will return in the next iteration, and a different set is removed. In this way, dropout decreases overfitting by preventing the network from becoming too dependent on any set of neurons. Once training is complete, all the neurons return. Dropout affects only neural networks during training.

Another significant innovation that benefits deep learning is Nesterov momentum (Sutskever et al., 2013). Momentum has been an important component of backpropagation training for some time. Polyak (1964) introduced the seminal momentum algorithm that is a regularization technique for gradient ascent/descent. Momentum backpropagation adds a portion of the previous iteration's weight change to the current iteration's weight change. Consequently, the weight updates have the necessary momentum to continue through local minima and to continue the descent to improve the loss function. Nesterov momentum (Nesterov, 1983) further enhances the momentum calculation and increases the effectiveness of SGD because SGD selects randomly sampled mini-batches from the data set for each iteration. Nesterov momentum decreases the likelihood of a particularly bad mini-batch from changing the weights into an irreparable state. Because of its demonstrated performance for deep neural networks, this dissertation will apply SGD to the neural network training.

Researchers have also utilized deep learning for recurrent neural networks. The research community has recently shown considerable interest in deep LSTM networks. Kalchbrenner, Danihelka, and Graves (2015) use a grid of LSTM units to achieve greater accuracy. Chung, Gulcehre, Cho, and Bengio (2015) introduced the gated recurrent network (GRU) and added an output gate, which allows greater accuracy as the time series increases in length. Unlike feedforward neural networks, LSTM and GRU are recurrent networks that can function as Turing machines (Graves et al., 2014).

Evolutionary Programming

Holland (1975) introduced evolutionary algorithms. Later, Deb (2001) extended this work to present genetic algorithms as they are known today. The genetic algorithm is a generic population-based metaheuristic technique to find solutions to many real-world search and optimization problems. Darwinian evolution inspired these algorithms. A population of potential solutions is evolved as the fittest population members produce subsequent generations through the genetic operators of crossover and mutation. Each of these potential solutions is referred to as either a genome or a chromosome (depending on the implementation). This evolutionary process is a search for the classic balance between exploitation and exploration. Mutation and crossover provide the genetic algorithm with the ability to explore and exploit the search space. Exploration occurs when the mutation genetic operator introduces randomness to the population. The crossover genetic operator exploits by creating new members containing traits from the best members of the population (Holland, 1975).

The genetic algorithm represents potential solutions as fixed-length vectors. This vector might be the weights of a neural network, coefficients of an expression, or any other fixed-length vector that must be optimized against an objective function. Mutation occurs by randomly perturbing the elements of a vector. Crossover is achieved by splicing together the vectors of two or more parent vectors.

An objective function serves to evaluate the population. The loss function of a neural network is somewhat similar to the evolutionary algorithm's objective function. Both the loss function and objective function provide a numeric value to be optimized. Some evolutionary algorithms also allow the objective function to be maximized. The choice between minimization and maximization depends on the domain of the problem. Although the loss function and objective function both accomplish similar goals, it is standard procedure to refer to the evaluation function for an evolutionary algorithm as an objective function.

While many problems can be modeled as a fixed-length vector, this representation is a limiting factor for classic genetic algorithms. They will never improve the underlying neural network algorithm even though they evolve the weights to produce better results. To evolve better algorithms, the computer programs themselves must become the genomes that will be evolved (Poli, Langdon, & McPhee, 2008). Genetic programming is an answer to the fixed-length issue of genetic algorithms.

Rather than evolving a fixed-length vector, genetic programming evolves representations of actual computer programs to achieve an optimal score to an objective function. Koza (1992) popularized this active area of research in order to automatically generate programs to solve specific problems. The majority of Koza's research represents the genetic programs as trees. Although most genetic programming research focused on tree representation (White et al., 2013), there are other representations of the genetic programs, such as grids and probabilistic structures (Wolfgang Banzhaf, Francone, Keller, & Nordin, 1998). This dissertation research will use only the tree representation of genetic programs.

A tree-based genetic program is implemented as a directed acyclic graph (DAG). The tree is composed of connected nodes. The tree starts with a parentless root node. It connects to other nodes and points to still more nodes. Interior nodes with at least one child form the tree. Terminal nodes without children also exist. Ultimately, the tree reaches all of the terminal nodes, which represents the variables and constants. In turn, the operators using the variables and constants are composed of the interior nodes. The following expression could be represented as a tree for genetic programming:

$$\frac{\boldsymbol{x}}{\mathbf{2}} - \mathbf{1} + \mathbf{3}\boldsymbol{\cos}(\boldsymbol{y}) \tag{6}$$

It is common in computer science to represent these expressions as trees. Lisp and Scheme are early programming languages that utilized similar tree representations called S-Expressions (Sussman, Abelson, & Sussman, 1983). Figure 7 shows the expression mentioned above as a tree:

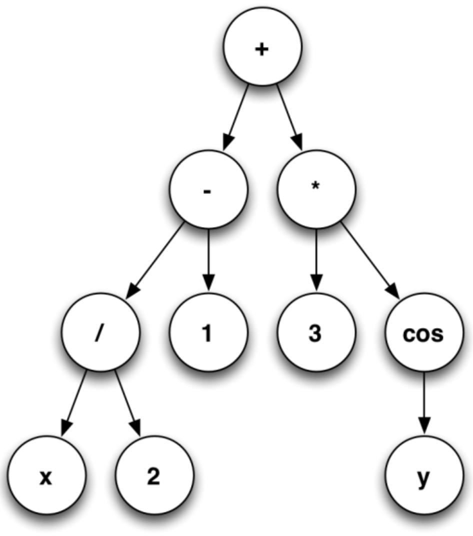


Figure 7. Expression tree for genetic programming

It is also possible to express entire computer programs as trees. The branching nature of a tree can encode if-statements and loops. The programs encoded into these types of trees can be Turing complete (Teller, 1994), which means they can theoretically compute anything (Turing, 1936). Additionally, nodes can be created to change the values of variables. However, trees are not the only representation for genetic programs.

Much of the research into genetic programming has involved the best way to represent the genetic programs (Poli et al., 2008).

All evolutionary algorithms must have processes for exploitation and exploration. For genetic programming, crossover and mutation can accomplish these functions. The representation of the underlying data dictates the exact nature of the mutation and crossover algorithms. Koza (1992) defined several possible algorithms for crossover and mutation of genetic programming trees. The two most popular are point crossover and subtree mutation. This dissertation includes point crossover and subtree mutation for its genetic programming solution. Figure 8 shows point crossover:

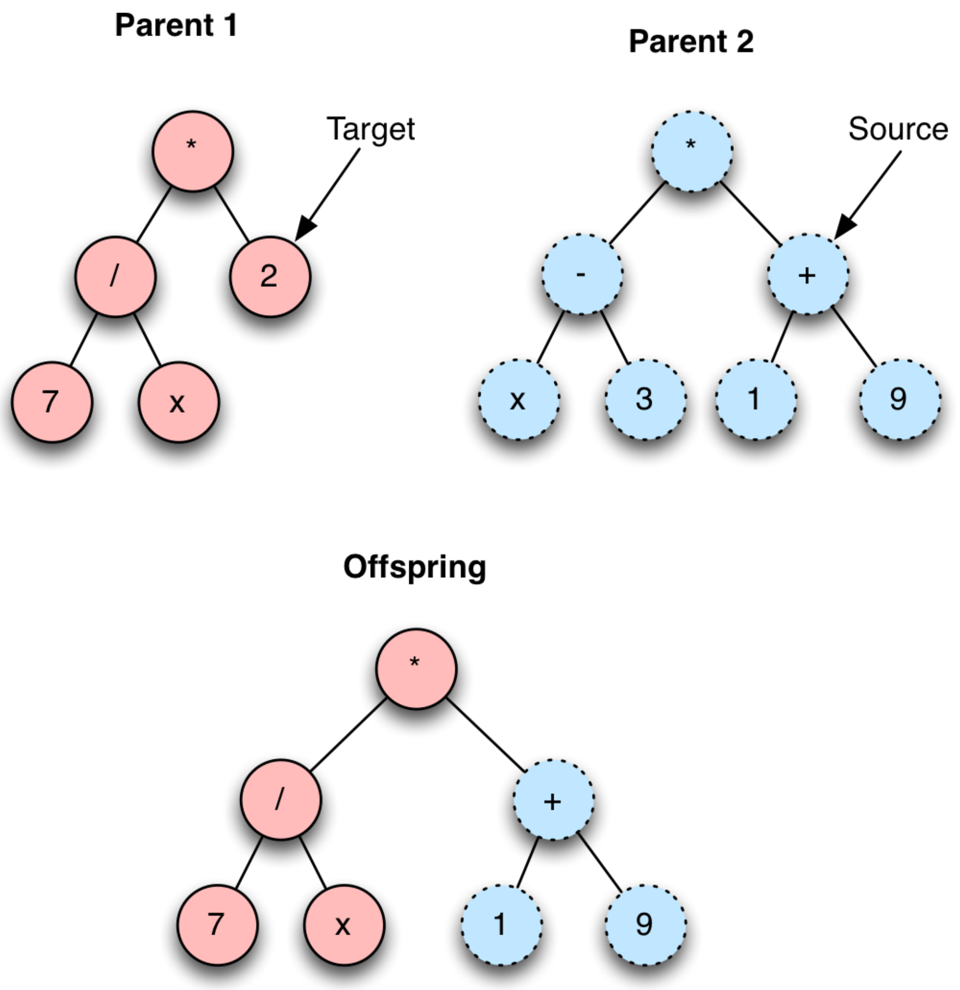


Figure 8. Point crossover

To implement point crossover, two parents are chosen. The algorithm chooses a random target node for parent 1 and a random source node for parent 2. Then it creates the offspring by cloning parent 1 and replacing the target node on the offspring with a copy of the subtree at the source node on parent 2. The crossover operation does not modify either parent, and the offspring may be more fit than the parents. Most importantly, crossover does not introduce new information; it only recombines existing information. As a result, crossover exploits rather than explores.

Koza also defined subtree mutation, as shown in Figure 9:

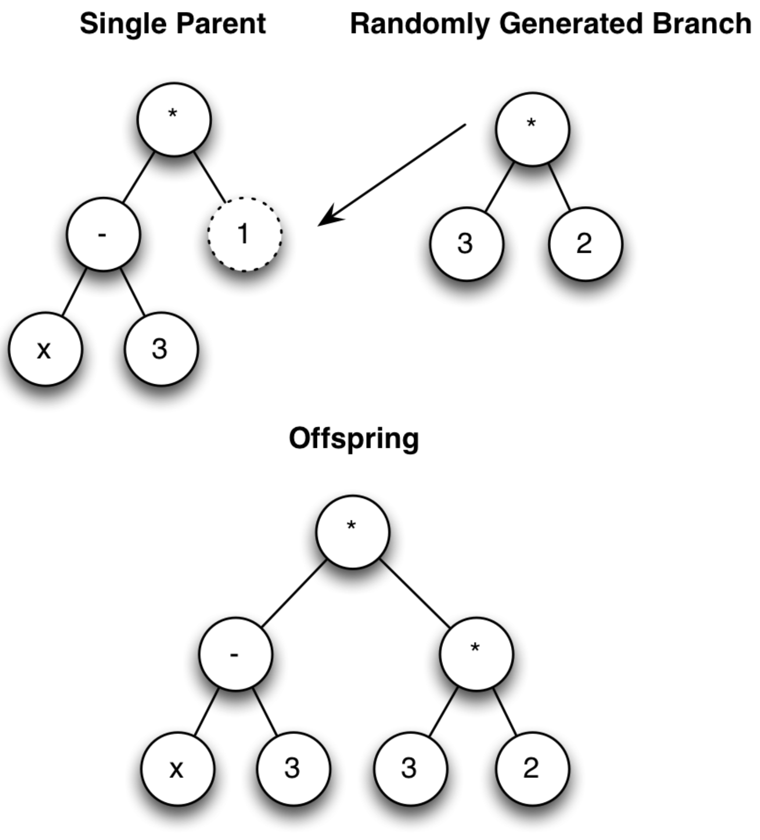


Figure 9. Subtree mutation

Mutation uses only a single parent, and, like crossover, the operation does not modify the parent. The algorithm chooses a random insertion node on the parent and generates a new random branch. Cloning the single parent and grafting the randomly generated branch onto the offspring at the previously chosen random insertion node creates the offspring. It is important to note that the mutation operator does add new information to the genome and, therefore, explores rather than exploits.

Speciation

It can be difficult to produce viable offspring from a crossover operation between two dissimilar genomes. This fact is true in real life as well. Two animals are considered to be from the same species if they are capable of producing offspring (Dawkins, 1976). In genetic programming, if the two parent genomes are similar, the probability increases for the crossover operator to produce offspring superior to the parents (Poli et al., 2008). Although there are several speciation strategies, this dissertation uses the strategy included with the Encog framework that is similar to the speciation algorithm of Stanley and Miikkulainen (2002) for the NEAT neural network. In that particular algorithm, the larger of the two parent trees is chosen, and the percentage of the same nodes of the larger tree that are also in the smaller tree is calculated. Genomes that have a percent similarity above a configurable threshold are considered to be in the same species. The species divisions are recalculated at the end of the training iterations. As a result, a child is not necessarily in the same species as the parents.

Other Genetic Program Representations

Trees are not the only means of representing genetic programs. Modern computers portray computer programs as a series of linear instructions (Knuth, 1997), not as trees.

Although these programs can be written as trees, it is often not practical because the linear nature of programming will create tall, unbalanced trees. This problem led a number of researchers to investigate a linear representation of the genetic programs. Poli et al. (2008), Wolfgang Banzhaf (1993), Perkis (1994), and Diplock (1998) sought to implement genetic programing in a fashion that mirrored the linear computer architecture. P. Nordin (1994), Peter Nordin, Banzhaf, and Francone (1999), Crepeau (1995), and Julian F Miller and Thomson (2000) went even further and evolved bit patterns that represented the actual CPU machine language instruction codes. The linear genetic programming systems create code that closely resembles pseudocode. As a result, a human programmer can more easily interpret linear genetic programs than a tree-based genetic program.

Cartesian Genetic Programming (CGP) represents the evolvable genetic programs as two-dimensional grids of nodes (Julian Francis Miller & Harding, 2008). CGP easily encodes computer programs, electronic circuits, neural networks, mathematical expressions, and other computational structures. Fixed-length vectors serve as integer-based grids for the crossover and mutation genetic operators.

Many genetic programs operate exclusively on floating point data. While some research into genetic programming have introduced Booleans, strings and other types, often only a single type is used. Such types are important; as traditional computer programs frequently use many different data types. If genetic programming is to evolve actual computer programs, it is important to offer this same multi-type flexibility. Unfortunately, data types such as *integer*, *string*, and *structure* complicate genetic programming because not every operator can accept all types. For example, the “-”

operator can easily be applied to integers and floating-point numbers, but it is undefined for strings or Booleans. To overcome this limitation, Worm and Chiu (2013) created a system of grammar rules that document dependencies between operators and data. For example, “-” works with numbers but not strings. These rules restrict the crossover and mutation operators and ensure that new programs are valid.

Genetic Programming for Automated Feature Engineering

Examples of the application of genetic programming to feature engineering exist in literature. Although this research is similar to the proposed dissertation, the method in this dissertation is unique. Specifically, this dissertation leverages unique characteristics of deep neural networks to constrain the infinite search space of potential engineered features. Additionally, the proposed algorithm introduces a novel objective function for genome selection and evaluates many potential engineered feature genomes simultaneously.

Guo et al. (2005) demonstrated that feature engineering could take advantage of genetic programming within the domain of fault classification. These researchers employed the Fisher criterion as the fitness function for the genetic program. This criterion is typically used in conjunction with Linear Discriminant Analysis (LDA) (Fisher, 1936). The criterion also measures the inter-class scatter between two classes in a classification problem. The work of Guo et al. experimented with expressions of original features in order to linearly separate pairs of classification outcomes in the training set. This approach differs from the proposed research because it does not attempt to exclude engineered features that the neural network can easily synthesize on its own. In other words, the research of Guo et al. is not specifically tailored to deep neural networks. Guo et al. evaluated the results of feature engineering using a shallow 14-neuron network and a support vector machine.

Another application of genetic programming to feature engineering was the Ph.D. dissertation of Neshatian (2010). This dissertation introduced a genetic programming-based feature engineering algorithm and identified that objective functions that utilize an underlying predictive algorithm are computationally expensive. Rather than directly target one specific predictive model, the Neshatian algorithm uses a *decision stump*, which is essentially a single node on a decision tree. This single stump learns a threshold value that separates two classes with minimal overlap. In this way, the approaches of both Guo et al. (2005) and Neshatian (2010) are similar. To see the combined approach, consider a linear separation between two iris species in the classic Iris data set (Fisher, 1936), shown in Figure 10.

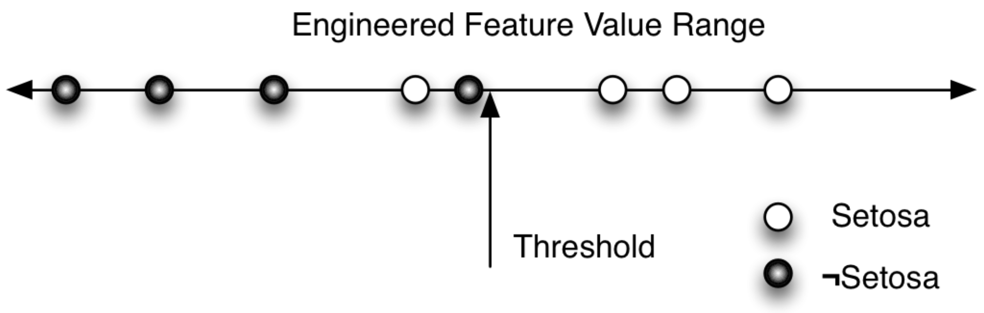


Figure 10. Feature engineering to linearly separate two classes

As can be seen, the genetic programming algorithm is seeking an engineered feature expression that isolates two of the classes (in this case iris species Setosa and the other two species). The commonly used iris data set contains the species and four measurements of 150 iris flowers of three species. The engineered feature becomes what is commonly referred to as a one-versus-rest model to distinguish one classification from the rest. Both Neshatian (2010) and Guo et al. (2005) demonstrated this model to benefit feature engineering for decision trees and shallow neural networks. However, Neshatian (2010) noted the inability of the algorithm to engineer features that enhanced neural network prediction, attributing this failing to the fact that neural networks synthesize comparable engineered features on their own. However, deep neural networks do benefit from feature engineering (Bengio, 2013; Blei et al., 2003; M. Brown & Lowe, 2003; Coates et al., 2011; Coates & Ng, 2012). Prior research also empirically demonstrated the benefits of feature engineering to deep neural networks (Heaton, 2016). As a result, the main purpose of this dissertation is to find these engineered features that increase the accuracy of deep neural networks.

Summary

Genetic programming and deep neural networks both accomplish similar tasks. They accept an input vector and produce either a numeric or categorical output. However, internally, each function differently. Deep learning adjusts large matrices of weights to produce the desired output. Genetic programming constructs computer programs to yield the desired output. A genetic program is considered Turing complete; however, a feedforward neural network is not generally Turing complete.

Feature engineering is a technique that preprocesses the data set to transform it into a new data set designed to better fit a model and to achieve better accuracy. Typically, feature engineering is a manual process. However, research interest in automating aspects of feature engineering exists. Because genetic programming is capable of evolving programs and expressions from its input features, it is logical to make it the basis for a feature engineering algorithm. Creating this algorithm is the goal of this dissertation. The exact methodology to build it will be discussed in the next chapter.

Chapter 3

Methodology

Introduction

The proposed research strives to build an algorithm capable of the automated creation of engineered features that increase the accuracy of a deep neural network. Unlike previous research that dealt with feature engineering, the proposed algorithm will allow its engineered features to draw upon multiple original values. It can also be considered as a theoretical infinite search over all combinations of the original feature set. The algorithm will return only combinations that enhance the learning of the deep neural network.

Instead of an infinite search, the proposed algorithm will perform a metaheuristic search. Because the search space is potential expressions, genetic programming is a natural choice of a metaheuristic. Other metaheuristic search algorithms, such as simulated annealing, Nelder-Mead (Nelder & Mead, 1965), particle swarm optimization (PSO) (Kennedy, 2010), or ant colony optimization (ACO) (Colorni, Dorigo, & Maniezzo, 1991), might be able to be used in connection with genetic programming to narrow the search space. However, the primary direction for this proposed research will be genetic programming.

Algorithm Contract and Specification

This research targets the type of deep neural network that is composed of the following standard components:

- Layered Feedforward Neural Network (Rumelhart et al., 1985)

- ReLU or Sigmoid Transfer Function for Hidden Layers (Glorot et al., 2011)
- Xavier Weight Initialization (Glorot & Bengio, 2010)
- Stochastic Gradient Descent (Bottou, 2012)

These algorithms will not be modified to achieve the goal of this research. The feature engineering algorithm produced by this research should enhance accuracy for a standard deep neural network, as configured above, that is supported by any common deep learning framework, such as Theano (Bastien et al., 2012; Bergstra et al., 2010), CNTK (D. Yu et al., 2014), TensorFlow (Abadi et al., 2016) or Encog (Heaton, 2015). However, the algorithm would not have wide application if its engineered features could only increase accuracy of non-standard implementations of deep learning.

The proposed algorithm accepts an operating specification encoded as a YAML file, which the following example shows:

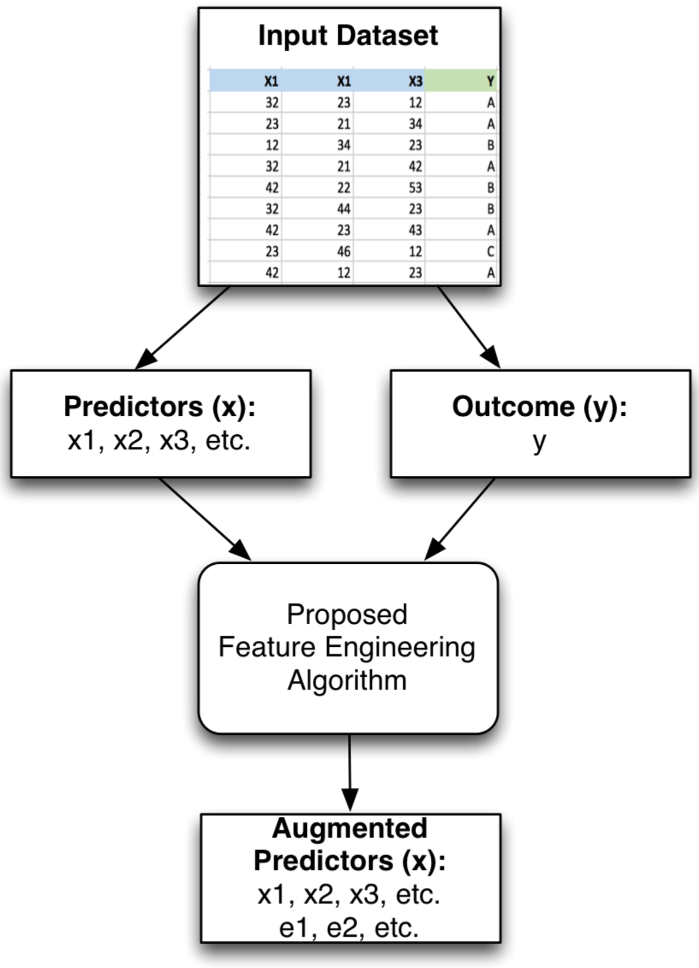
```
input_dataset: input.csv
input_x: [ 'acceleration', 'horsepower', 'cylinders'
input_y: [ 'mpg' ]
input_headers: true
output_features: features.csv
output_augmented_dataset: augmented_input.csv
prediction_type: regression
transfer_hidden: relu
...other configuration settings...
```

The above file specifies the configuration to analyze the standard UCI AutoMPG data set. The features *acceleration*, *horsepower* and *cylinders* allow the model to make a prediction. In this case, the prediction is *mpg*. The features and prediction are all column names in the original file.

The following definitions are for the individual configuration items:

- **input_data set:** The name of the data set that contains the predictors (*x*) and expected outcomes (*y*). The algorithm will use only the specified columns, except the target column(s) as the input.
- **input_x:** The list of column names to use as predictors (*x*). The names can be either symbolic or the zero-based index of the column.
- **input_y:** The list of column names to use as the target(s) (*y*). The names can be either symbolic or the zero-based index of the column.
- **input_headers:** Boolean to indicate if the input CSV has headers. If there are no headers, the columns must be referenced by their zero-based index.
- **output_features:** A CSV file that will contain a summary of the engineered features.
- **output_augmented_dataset:** The output CSV file that contains the input CSV file data, along with the new engineered features.
- **prediction_type:** The type of prediction and will be either classification or regression.
- **transfer_hidden:** The type of hidden transfer function to target. It should be either *relu, sigmoid* or *htan.*

As the algorithm develops, it will add more configuration items. The algorithm's input file is the same input file that would train a neural network. The output file will have appended engineered feature columns. This output file could train a neural network. All common neural network frameworks can train from files of this format. The goal of this research is to show that a neural network trained from the algorithm's augmented input will produce a neural network that will allow a more accurate neural network for some data sets. Figure 11 summarizes this high level design:



X1	X1	X3	Y
32	23	12	A
23	21	34	A
12	34	23	B
32	21	42	A
42	22	53	B
32	44	23	B
42	23	43	A
23	46	12	C
42	12	23	A

Figure 11. Algorithm high level design and contract

Algorithm Design Scope

The proposed algorithm will utilize the genetic programming components of Encog, which implement a genetic programming framework according to Koza (1992). The software developed to achieve the research proposed will center primarily on the following aspects of genetic programming:

- Constraints to narrow the search domain
- Objective function to effectively evaluate candidate features
- Post processing analysis of genetic population

The following sections highlight the approach for each of these components:

Narrowing the Search Domain

Placing search constraints upon a genetic programming algorithm is possible (Gruau, 1996; Janikow, 1996). These limits are sometimes implemented as an additional objective, producing a multi-objective genetic program. The primary objective for a genetic programming algorithm is to achieve a favorable score from the deep learning loss function. In prior research, it was determined that certain classes of transformations, such as simple power functions, ratios, differences and counts do not benefit deep neural networks (Heaton, 2016). Therefore, a second objective for a genetic algorithm is to avoid evaluating these transformations. This dissertation will expand on this previous work that determines undesirable expressions so that the proposed algorithm can focus on the most beneficial structures for deep learning.

Genetic programming requires a palette of operators from which to construct expressions. The selection of these operators is critical to a good solution. For example, some problems may benefit by adding the trigonometry functions. However, the trigonometry functions might be unnecessary for other problems. Sometimes problem specific functions are added to the operator set for genetic programming. For the purposes of this research, the operator palette will be small; however, some functions that are specific to feature engineering will be added. The following palette of operators are planned:

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)
- Mean of Column/Feature
- Standard Deviation of Column/Feature
- t-SNE Distance to Centroid of Each Class

The final operator described above uses the dimensional reduction algorithm t-SNE (Van der Maaten & Hinton, 2008) to cut the dimensions of the feature vector to 3. It also calculates the distance between the feature vector for the current data set item and the centroid of each class in a classification problem. If the data set is for regression, then the t-SNE operator will not be available. Other operators not included above may be added as this research is conducted.

Neural networks are calculated by applying a weighted sum to each neuron, as demonstrated by Equation 1. Examining the equation reveals that neural networks inherently have the ability to multiply and sum. As a result, neural networks do not tend to benefit as much from engineered features involving simple multiplication and addition (Heaton, 2016). Therefore, it might not be worth the time to discover an engineered feature as complex as the following equation:

$$\boldsymbol{f_e = \frac{3f_1f_2}{2f_3}} \tag{7}$$

In the above equation, an engineered feature (f_e) was calculated using three of the original features. Because neural networks can perform their own multiplication, it might be possible to simplify to the following equation:

$$\boldsymbol{f_e} = \frac{\boldsymbol{f_1 f_2}}{\boldsymbol{f_3}} \tag{8}$$

The previous two equations are not mathematically equivalent; however, the second equation might be sufficient because the neural network would have the ability to multiply 3/2 by the engineered feature. Since the proposed research is targeted at deep learning, engineering parts of the feature that the neural network can easily learn during training should be avoided.

Creating an Efficient Objective Function

An objective function is needed to guide the genetic selection operator because the research utilizes a genetic programming algorithm. The experiments will use a multi-objective function (Deb, 2001) that will balance between finding effective engineered features and avoiding engineered features that are known to be ineffective for deep neural networks.

The first objective is to evaluate the value of engineered features. To accomplish this task, a control neural network will first be trained with just the original features present. Training of the control neural network will need to be performed only once at the start of the process. A potential engineered feature will be added to the feature vector and to a new neural network so that it can be evaluated. The difference between the control neural network's loss function and the engineered neural network's loss function will become the score returned by the objective function. Scores above 0 do not improve the neural network's predictive power; scores below 0 improve the neural network. The genetic programming algorithm will strive to attain scores below 0.

It will be critical to optimize the computational performance of the loss function. There are numerous opportunities to optimize this function. One potential optimization is to decrease the size of the deep neural network and the number of training iterations, thereby decreasing training effectiveness. However, as long as both the control and test neural network receive equal treatment, the results should indicate if the engineered features are improving the neural network loss function result. It is important to remember that the goal of the objective function is not to fully train the neural network. Instead, the goal is to gain some indication of the effectiveness of the engineered features. Other novel techniques will be evaluated to produce an objective function with acceptable performance.

The second objective will be to avoid the expressions that are known to not improve a deep neural network's effectiveness (Heaton, 2016). Earlier research determined that counts, differences, logs, power functions, rational polynomials, and radicals were not particularly effective engineered features for deep learning. Similarly, earlier research showed that deep neural networks benefited from polynomial, ratio, and rational difference features. Engineered features that resemble polynomial, ratio, and rational difference will receive a score bonus.

Experimental Design

To overcome the issues and barriers previously mentioned, a series of experiments will be conducted. The results from the first four experiments will identify the design characteristics of the genetic programming elements identified in the previous section. Specifically, these experiments will show how to design constraints, measure the success of engineered features, and analyze the population. The proposed experiments are listed here:

- Experiment 1: Limiting the Search Space

- Experiment 2: Establishing Baseline
- Experiment 3: Genetic Ensembles
- Experiment 4: Population Analysis
- Experiment 5: Objective Function Design
- Experiment 6: Automated Feature Engineering

Each of these experiments will be measured as described in the next section.

Measures

Neural networks are a stochastic model, and measuring the relative performance of the individual training runs can be difficult. This randomness first occurs because neural networks are initialized with random weights. The randomness continues with the stochastic gradient descent-training algorithm that chooses randomly sampled mini-batches to train the neural network. This randomness in the training process prevents two neural network training runs from producing the same training result.

Bagging (Breiman, 1996) is an ensemble technique that turns the randomness of the neural network into a strength. Bagging trains many neural networks and averages the results together. All measures of neural networks performed by this dissertation will be bagged in a method similar to that of Prechelt (1994). If bagging does not provide the stability to perform accurate measures, then dropout may also be considered. Dropout is related to bagging in that each set of dropped out neurons can be considered a bagging cycle. Measurement of both feature importance and neural network accuracy will use either bagging or dropout. No matter the technique chosen, it will be applied across all measures performed in this research.

Two primary metrics will be considered for this dissertation: feature importance and network error. Feature importance determines how important an individual member of the feature vector is to the predictions of the neural network. Neural network error, the second metric, determines the accuracy of a neural network. Both metrics are described in this section.

Feature importance can be measured for a model in two different ways: model agnostic and model dependent. A model agnostic calculation measures feature importance purely by querying the model. Internal analysis of the model is not needed. A model agnostic metric will work with any type of model, including a neural network, random forest, or support vector machine (SVM). Permutation feature importance (Breiman, 2001) is a model agnostic calculation technique that this research will investigate. Additionally, the model dependent approach of analyzing the weights of the neural network will also be considered. The choice of approach will remain as a configuration option. Results will be reported on the approach that is the most consistent measure of the importance of the neural network features.

It is also necessary to measure the accuracy of the neural network. The proposed research will use the RMSE and multi-class log loss error functions. RMSE will be the error function for all regression problems, and multi-class log loss will be calculated for classification. RMSE (McKinney, 2012) and log loss are given in the following equations:

$$\boldsymbol{RMSE} = \sqrt{\frac{\sum_{i=1}^{n\|y\|}(\hat{y}_i - y_i)^2}{\|y\|}} \tag{9}$$

$$\boldsymbol{MLogLoss} = -\frac{\mathbf{1}}{\|\boldsymbol{y}\|}\sum_{\boldsymbol{i=1}}^{\boldsymbol{N}}\sum_{\boldsymbol{j=1}}^{\|\boldsymbol{y}\|}\boldsymbol{y_{i,j}}\mathbf{log}(\boldsymbol{\hat{y}_{i,j}}) \tag{10}$$

For both equations, *N* represents the number of training set elements. The vector *y-hat* represents the actual output vector from the neural network, and the vector *y* represents the expected output from the neural network.

Whether measuring accuracy or feature importance, bagging or dropout will be employed, and the lowest metric will be reported. The presentation format for the results of each experiment will be provided in the next sections.

Experiment 1: Limiting the Search Space

The first experiment continues feature engineering research performed prior to this dissertation (Heaton, 2016). This earlier research demonstrated that neural networks, decision trees, random forests, and gradient boosting machines benefited from different types of engineered feature. Additionally, the research showed that there were several types of engineered feature that were not particularly easy for a neural network to learn on its own. If a neural network can learn to synthesize an engineered feature on its own, adding this feature will not be particularly helpful to the neural network.

The search space of the proposed genetic programming-based algorithm can be narrowed down. The process entails constraining the genomes to expressions similar to expressions that are known to be difficult for the neural networks to synthesize. Heaton (2016) shows the effectiveness of a deep neural network learning to synthesize several types of engineered feature. The neural network learned single-feature transformations, such as log, polynomial, rational polynomial, power, quadratic, and square root without problems. However, multi-featured transformations were not as simple. The neural network could calculate counts and differences. Rational difference and simple ratios were more difficult for the neural network.

Experimental Design

The first experiment will examine the effectiveness of other expression types with neural networks using the same technique as a previous investigation (Heaton, 2016). This method for testing an expression involves generating a data set that uses an outcome that is the result of the expression being tested as well as randomly sampled values for the input. If the neural network converges to a low RMSE error, then the neural network can learn the expression. Furthermore, engineered features in the same format as that expression are more likely to produce more accurate neural networks.

Results Format

The results from this experiment will be reported in tabular format with the following columns: expression, name, RMSE/Log Loss, and genetic program RMSE. Consistent with Heaton (2016), the lowest RMSE/Log Loss of five training processes will be reported. The result format from this experiment will appear similar to Table 2.

Table 2. Experiment 1 results format, GP vs. neural network

#	Name	Expression	RMSE/ Log Loss	GP RMSE/ Log Loss
1-1	Ratio	$\frac{x_1}{x_2}$	###	###
1-2	Ratio Difference	$\frac{x_1 - x_2}{x_3 - x_4}$	###	###
…	…	…	…	…

Genetic programs can be taught to approximate expressions, just like neural networks. While the original research did not evaluate genetic programming, seeing the error for the genetic program might give an indication of the ability of genetic programming to synthesize the expressions. It is expected that genetic programming will easily synthesize any of the expressions and achieve a low error. If genetic programming does not achieve a low error, the hyper-parameters for the genetic programming algorithm will be tuned.

Experiment 2: Establishing Baseline

For the dissertation algorithm to be effective, engineered features from this algorithm must enhance neural network accuracy. To measure this performance, a number of public and synthetic data sets will evaluate it. It is necessary to collect a baseline RMSE or log loss of a deep neural network with that data set that receives no help from the dissertation algorithm. The neural network topology, or hyper-parameters, will be determined experimentally. It is important that the topology include enough hidden neurons that the data set can be learned with reasonable accuracy.

Experimental Design

The baseline experiment will provide a neural network result to compare those from the dissertation algorithm. Not all data sets will benefit from feature engineering, so it is important to select a number of real world and synthetic data sets. The initial collection of data sets is listed later in this chapter. Additional data sets will be added or generated if needed to find a data set that demonstrates improvement from engineered features.

Each neural network will be trained until the validation error no longer improves. Several runs will be conducted, and the lowest error will be reported. Bagging or dropout will be conducted to get a constant result, as previously described. Training parameters and neural network topology will be set as defined in the previous section, “Measures.”

Results Format

The results from this experiment will be reported in tabular format with the following columns: data set number, name, network topology, RMSE/Log Loss. The result format from this experiment will appear similar to Table 3.

Table 3. Experiment 2 result format, neural network baseline

#	Name	Topology	RMSE/Log Loss
2-1	Auto MPG	Regression: 5:20:15:3	###
2-2	Wisconsin Breast Cancer	Classification 15:20:15:1	###
...	...	...	...

Experiment 3: Genetic Ensembles

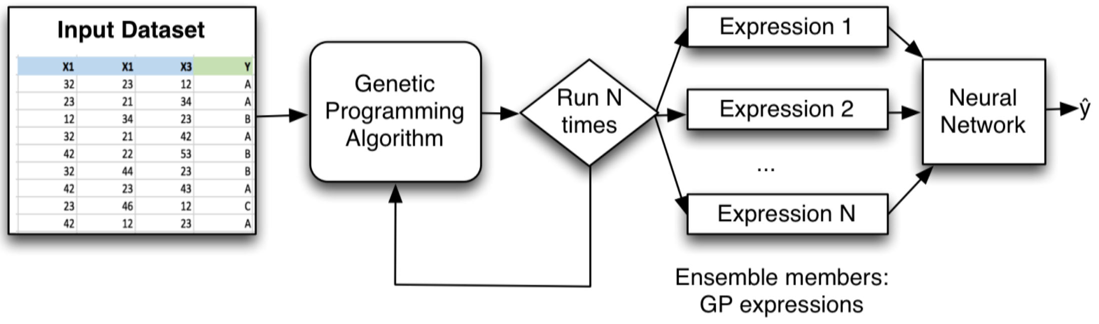
Neural networks and genetic programs can both function as regression and classifier models. It is possible to combine models that are trained to accomplish the same objective into ensembles (Dietterich, 2000). A simple blending ensemble takes the output from several models and uses another model type such as a neural network, generalized linear model (GLM), or simple average to combine the results of the member models.

Model ensembles work best when the member models are orthogonal. In other words, they do not always produce the same decision (Dietterich, 2000). In some ways, the models can be considered complex engineered features that are fed into the model that makes the final decision. Further experiments will refine these complex genetic program models into features that can augment the original feature vector.

Experimental Design

Experiment 3 parallels Experiment 2 to some degree. However, instead of evaluating individual neural networks, ensembles of neural networks and genetic programs are evaluated for each of the data sets. If the score of a particular data set is improved from neural network to ensemble, there is a chance that a further refined genetic programing engineered feature from this dissertation might be able to improve the neural networks accuracy.

Two different ensemble architectures will be considered for this experiment. The first ensemble, shown in Figure 12, uses a neural network as the ensemble fed by *N* genetic program expressions. The second ensemble, shown in Figure 13, uses a combination of one neural network (from experiment 1) and *N* genetic program expressions to form an ensemble that is aggregated by either a generalized regression model (GLM) or a linear regression ensemble.



X1	X1	X3	Y
32	23	12	A
23	21	34	A
12	34	23	B
32	21	42	A
42	22	53	B
32	44	23	B
42	23	43	A
23	46	12	C
42	12	23	A

Figure 12. Neural network genetic program ensemble

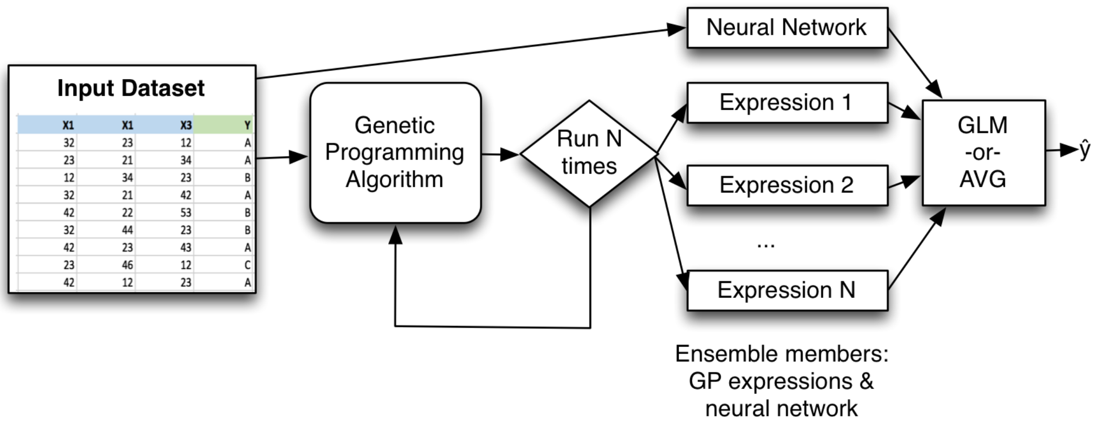


Figure 13. AVG/GLM neural network genetic program ensemble

In both cases, the inputs to the ensemble member models are those from the original feature vector. The inputs to the ensemble component (neural network, GLM or AVG) are the single-valued outputs from the ensemble members. As a result, the ensemble component will have a number of inputs equal to the number of ensemble members. The output from the ensemble component ($\hat{y}$) is considered to be the prediction of the ensemble model, and it is the value that allows the accuracy of the ensemble's prediction to be reported.

To determine whether to use a GLM or AVG as the ensemble, it is necessary to evaluate its performance for a single classification and regression data set. The results will be reported only for the chosen ensemble algorithm. Additionally, the program will also report the results for the data sets that formed the basis for the ensemble algorithm.

Results Format

The results from this experiment will be reported in tabular format with the following columns: data set number, name, and the RMSE/Log Loss for the neural network ensemble, as well as the AVG or GLM ensemble. For each of the two ensembles, the RMSE/Log Loss will be reported for 3, 5, and 10 ensemble members (*N*). The result format from this experiment will appear similar to Table 4.

Table 4. Experiment 3-result format, neural network genetic program ensemble

		Neural Network Ensemble			AVG/GLM Ensemble		
#	Name	N=3	N=5	N=10	N=3	N=5	N=10
3-1	Auto MPG	##	##	##	##	##	##
3-2	Wisconsin Breast Cancer	##	##	##	##	##	##
...	...	...	...	...	...	...	...

This experiment will indicate early if genetic programming can automatically synthesize information to the data set that the neural network alone could not determine. The results of Experiment 3 can be compared to Experiment 2. For the data sets where the ensemble performed better than the neural network alone, there is hope that genetic programming can engineer a viable feature. If Experiment 3 receives a better result on the ensemble than Experiment 2 did with a neural network alone, a viable feature has already been engineered.

The comparative analysis of Experiments 2 and 3 will be reported in tabular form, similar to Table 5.

Table 5. Experiments 2 & 3 comparative analysis format

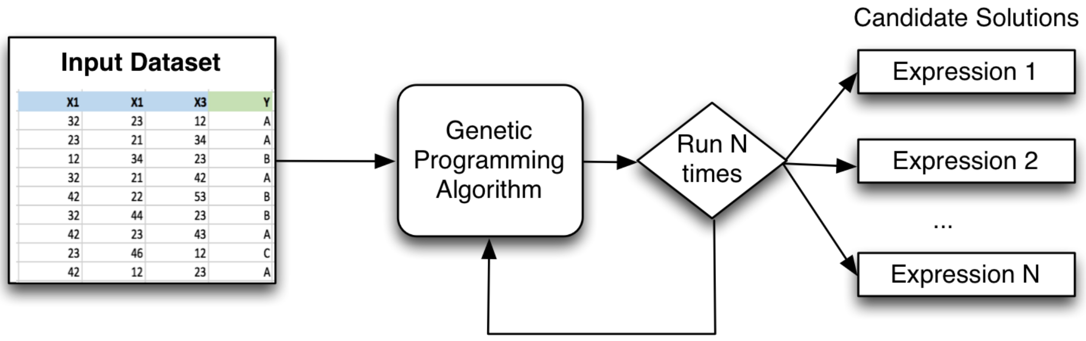
#	Name	Experiment 1 RMSE/ Log Loss	Experiment 2 Best RMSE/ Log Loss	Best
3b-1	Auto MPG	###	###	Experiment 2
3b-1	Wisconsin Breast Cancer	###	###	Experiment 1
...	...	...	...	...

Experiment 4: Population Analysis

Previous experiments fit genetic programs to function as complete models in their own right. These complete models might use the entire feature vector and be complex. Features that are engineered by human analysis are usually simple combinations of a handful of the original feature vector. The purpose of this experiment is to determine if the complex genetic programming models can be distilled to a representation that might give insight on how to combine elements of the original feature vector into viable, engineered features. These simpler features might perform enough of the calculation to help augment the data for a neural network.

Experimental Design

The input data set will allow the generation of several candidate solutions. This process to generate the ensemble members will be the same as in Experiment 3. Figure 14 shows the overall flow for generating the candidate solutions.



X1	X1	X3	Y
32	23	12	A
23	21	34	A
12	34	23	B
32	21	42	A
42	22	53	B
32	44	23	B
42	23	43	A
23	46	12	C
42	12	23	A

Figure 14. Generate candidate solutions

Unlike Experiment 3, the value of *N* will be larger. More candidate solutions create more data from which the program can find a pattern. The program searches for the pattern of pairs of input features that are often connected by a common operator, higher in the tree. Figure 15 shows a hypothetical example of a pattern.

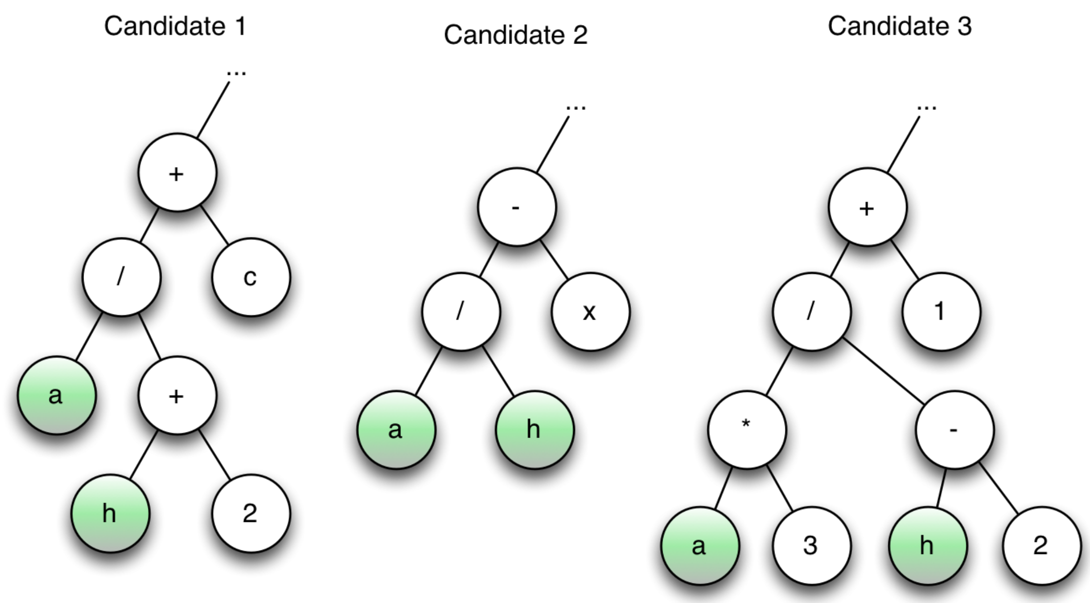


Figure 15. Branches with common structures

This figure shows three branches that might be present on trees from a much larger forest. The three branches always have the features *a* and *h* united by a division operator higher in the branch. Of course, these branches are part of complete trees in the forest of candidate solutions. Partial branches are shown for clarity and brevity. The fact that *a* and *h* often occur together in a ratio might indicate that final engineered features use these two in a ratio. This information could allow the construction of the features outright. It could also narrow the search space for a given data set.

Results Format

The results from this experiment will be reported in tabular format with the following columns: data set number, name, and the patterns found. Patterns will be reported in a format such as “a/b”, “a*b”, or “(a+b)/c”. The result format from this experiment will appear similar to Table 6.

Table 6. Experiment 4 results format, patterns in genetic programs

#	Name	Patterns
4-1	Auto MPG	(x1/x2)
		(x2*x3)
		((x1*x2)+x1)
4-2	Wisconsin Breast Cancer	(x2*x3)
...	…	...

Experiment 5: Objective Function Design

Evolutionary algorithms work through a selection process that is based on natural selection (Holland, 1975). To select from among the genomes, it is necessary to evaluate the effectiveness of the candidate-engineered features relative to each other. Several algorithms determine the feature importance of the neural network. For this research, a feature-ranking algorithm should be relatively stable, despite the stochastic nature of neural networks. Feature ranking stability will be measured by the degree to which multiple neural networks produce the same ranking of features for the same data set.

The goal of this experiment is to determine an effective feature ranking algorithm for the proposed dissertation research. The two main considerations for feature ranking are the following:

- Which feature-ranking algorithm is the most stable?
- How far should a neural network be trained?

It is desirable to keep the objective function as computationally inexpensive as possible. The faster the objective function executes, the larger a search space to be covered. To evaluate feature ranking, it is not necessary to train the neural networks to their maximum accuracy. Instead, it is crucial to train the neural network to the point that the ranking algorithm can give a stable assessment of the relative importance of each of the features. This experiment will strive to provide answers to both of these concerns.

Experimental Design

This experiment will evaluate the ranking of the original feature vector for each of the data sets considered for this dissertation. Each data set will be trained multiple times with its feature ranking evaluated at each training iteration. A validation holdout set will allow the training algorithm to know the stopping point. Once the validation set's error no longer improves, training will stop. The percent validation improvement will be reported for the iteration where the feature ranking first stabilized to the same order as the final feature ranking. The idea is that training could be stopped at the point that the validation error improvement hits this point, and a reasonably accurate feature rank could be determined.

Results Format

Each of the data sets will be evaluated, and the minimum validation set improvement will be reported. Additionally, the final feature rank will also be reported. The results will appear similar to Table 7.

Table 7. Experiment 5 results format, evaluating feature ranking

#	Name	Rank	Weight	Permutation
5-1	Auto MPG	x3,x2,x1,x10,x11	###	###
5-2	Wisconsin Breast Cancer	x3,x2,x4,x10,x1	###	###
...	...	...	...	...

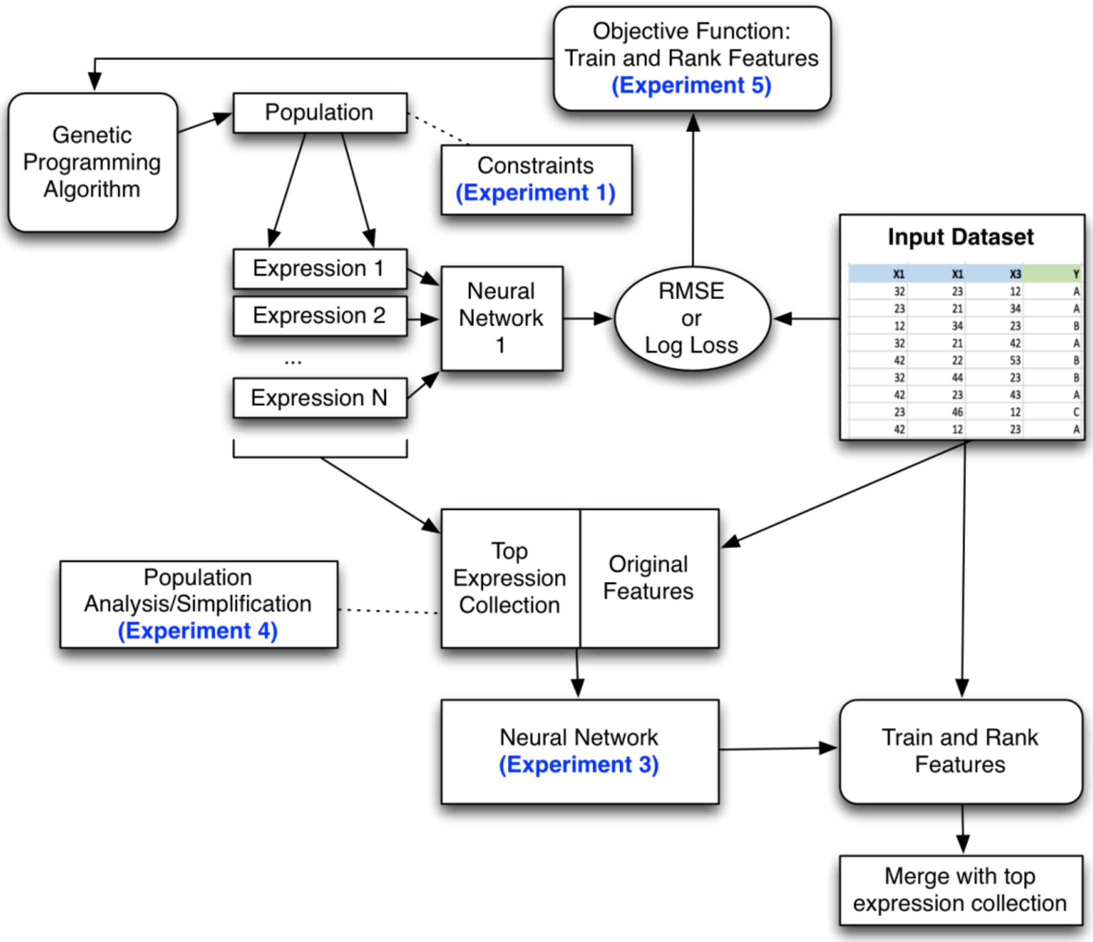
Experiment 6: Automated Feature Engineering

The final planned experiment brings the results of the previous five into the automatic feature-engineering algorithm, thereby accomplishing the objective of this dissertation. The goal of Experiment 6 is to compare each data set. It will show the neural network validation error next to the same neural network that has been augmented with features that the dissertation algorithm engineered.

Experimental Design

This experiment will be conducted similarly to Experiment 1, except that the neural networks will be augmented with features that were engineered from each of the data sets. The dissertation algorithm's effectiveness will be evaluated by comparing the neural network's accuracy on the augmented data set with the baseline result in experiment 6. Comparing the results of the dissertation algorithm to the Experiment 1 baseline should show that some of the data sets will have improved results.

The feature-engineering algorithm cannot be specified exactly at this point because the experiments 1, 3, 4, and 5 will guide its development. Experiment 2 establishes a baseline to compare the algorithm to. Additionally, Experiment 6 may undergo several iterations as the algorithm is enhanced and its performance measured. The final algorithm, measured by Experiment 6, might look similar to Figure 16.



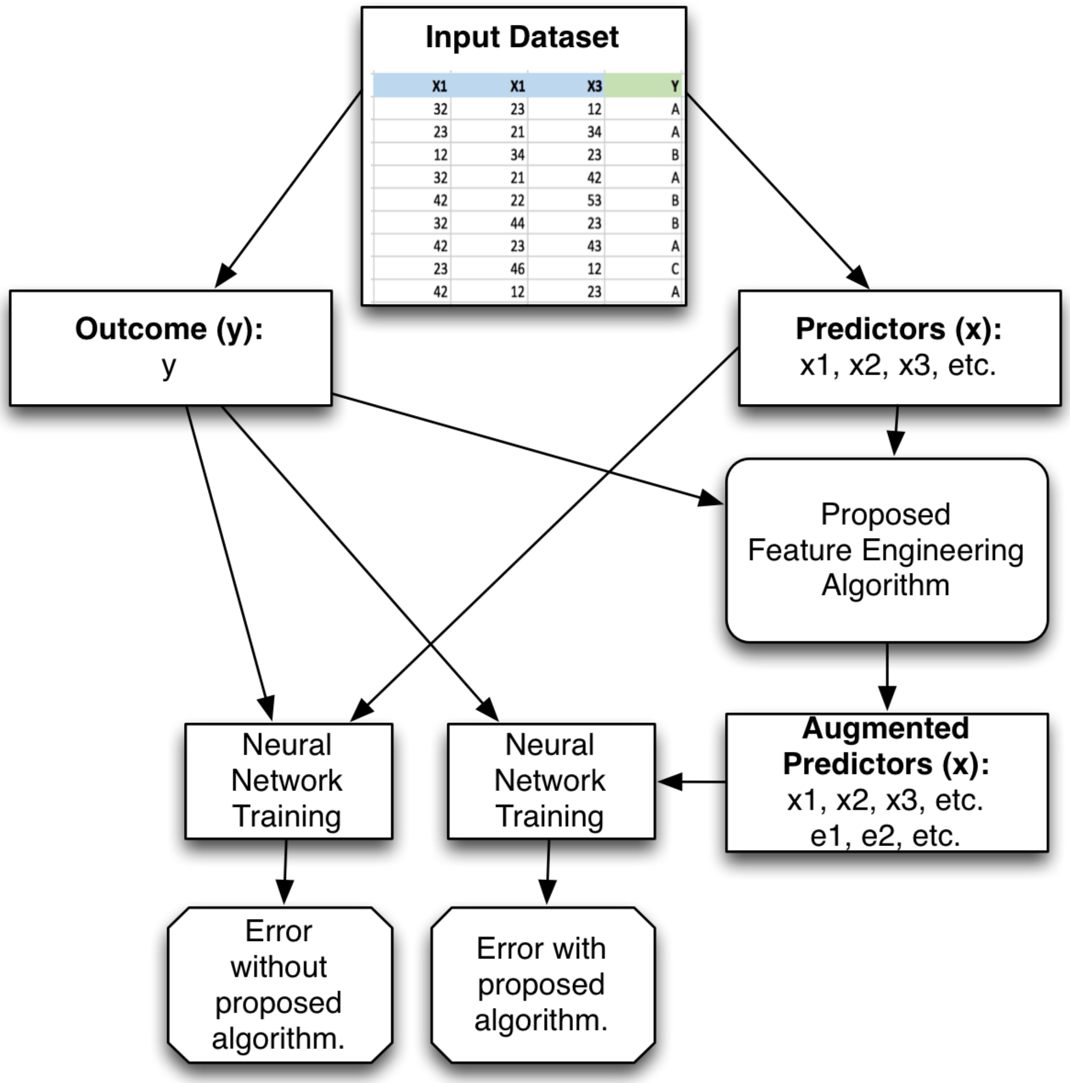
X1	X1	X3	Y
32	23	12	A
23	21	34	A
12	34	23	B
32	21	42	A
42	22	53	B
32	44	23	B
42	23	43	A
23	46	12	C
42	12	23	A

Figure 16. High-level overview of proposed algorithm

This algorithm uses two neural networks. Neural Network 1 evaluates a population of potential engineered features. This population will be constrained using information learned from Experiment 1. The objective function comes directly from Experiment 6. Experiment 4 will provide information to refine the candidate expression from the population into potential engineered features. These engineered features will be ranked and evaluated using Neural Network 2 and examined to join a list of top engineered expressions.

The above flow chart will likely change to some degree as the experiments are evaluated and a system is built to automatically engineer features. The goal will remain to engineer features that are able to boost the baseline performance (Experiment 1) of the data sets.

Two neural networks are trained. The first neural network will be trained on the original predictors (*x*) and the second on the augmented feature vector. Both are trained to produce the outcome (*y*) that this algorithm does not alter. Figure 17 shows the setup for the final evaluation.



X1	X1	X3	Y
32	23	12	A
23	21	34	A
12	34	23	B
32	21	42	A
42	22	53	B
32	44	23	B
42	23	43	A
23	46	12	C
42	12	23	A

Figure 17. Dissertation algorithm evaluation

This figure closely resembles Figure 11. Essentially, the program forks the input data set to be fed to two neural networks for training. The leftmost neural network trains without the proposed algorithm, and the neural network on the right trains with the proposed algorithm. The objective is to determine if the neural network with augmented predictors can perform better than a neural network with only the original features.

Results Format

The results of this experiment will show which features could be engineered for each of the data sets in the dissertation. The results will contain the following columns: experiment number, data set name, engineered features employed, neural network error (RMSE or log loss), augmented neural network error (RMSE or log loss), and change by using algorithm. The neural network error column will report exactly the same values as Experiment 1 because this experiment compares the dissertation algorithm to the neural network baseline. Table 8 shows the anticipated format for the results of Experiment 6:

Table 8. Experiment 6 results format, engineered feature effectiveness

#	Name	Engineered Features	Baseline Error	Augmented Error	Error Change
6-1		#### ####	####	####	####
6-2		#### ####	####	####	####
...	...	...	...	...	...

It is assumed that the algorithm will not find an engineered feature set for each data set. The number of data sets that a viable engineered feature and the degree to which the error of the neural network decreases will be important factors for measuring the usefulness of the algorithm developed.

Real World Data sets

The success of an automatic feature-engineering algorithm will ultimately be measured by its success in the generation of actual features for a real world data set. The following will be the primary sources for real world data sets:

- UCI Machine Learning Repository (Newman & Merz, 1998)

- PROBEN1 Datasets (Prechelt, 1994)
- Kaggle Competitions

Several data sets will come from the UCI machine learning set. Time will be needed to standardize each of these data sets in this dissertation. Data sets that have the following attributes will be favored:

- No image or audio data sets
- At least 10 numeric (continuous) features
- Features should be named, such as measurements, money or counts

The following five UCI data sets appear to be good candidates for this research:

- Adult data set
- Wine data set
- Car evaluation data set
- Wine quality data set
- Forest fires

Other UCI data sets will be considered, as needed, for this research.

Prechelt (1994) introduced the PROBEN1 collection of data sets. This collection contains 13 standardized data sets. The paper that presented PROBEN1 has neural network benchmark results. Additionally, the PROBEN1 paper has over 900 citations, many of which publish additional neural network results on these data sets. These characteristics make the PROBEN1 data sets good candidates for comparison in this study.

Kaggle data sets may also be considered for benchmarking this dissertation. Many Kaggle data sets are compatible with the previously stated desired characteristics of data sets for this dissertation research. Additionally, because Kaggle is an open competition, there are numerous published results for a variety of modeling techniques.

Synthetic Data sets

Not all data sets will see increased accuracy from engineered features, especially if the underlying data do not contain relationships that feature engineering can expose. As a result, it will be necessary to create data sets that are designed to include features that are known to benefit deep neural networks. The feature-engineering algorithm in this research will be tested to see if it is capable of finding the engineered features that are known to help neural networks predict these generated data sets.

The program will generate data sets that contain outcomes that are designed to benefit from feature engineering of varying degrees of complexity. It is necessary to choose engineered features that the deep neural networks cannot easily learn for themselves. The goal is to engineer features that help the deep neural network—not features that would have been trivial for the network to learn on its own. In previous research Heaton (2016) formulated a simple way to learn the types of features that benefit a deep neural network was devised. Training sets were generated in which the expected output was the output of the engineered feature. If the model can learn to synthesize the output of the engineered feature, then adding this feature will not benefit the neural network. This process is similar to the common neural network example of teaching itself to become an XOR operator. Because neural networks can easily learn to perform as XOR operators, the XOR operation between any two original features would not make a relevant engineered feature.

Resources

The hardware and software components necessary for this dissertation are all standard, readily available, common, and off-the-shelf personal computer system components and software. Two quadcore Intel I7 Broadwell-equipped machines with 16 gigabytes of RAM each are available for this research. These systems will perform the majority of computations needed to support this research. If additional processing power is required Amazon AWS virtual machines will be used.

The Java programming language (Arnold, Gosling, & Holmes, 1996) will serve as the programming language to complete this research. The Java 8 version (JDK 1.8) will provide the specific implementation of the programming language. In addition, Python 3.5 (Van Rossum, 1995) will work in conjunction with Scipy (Jones, Oliphant, Peterson, & al., 2001), scikit-learn (Pedregosa et al., 2011), TensorFlow (Abadi et al., 2016) for deep learning. The Python machine learning packages will be useful to compare select neural networks and feature combinations with the Encog library.

Encog version 3.3 (Heaton, 2015) will provide the deep learning and genetic programming portions of this research. Encog provides extensive support for both deep learning and genetic programming. Additionally, Encog is available for both the Java and C# platforms. The author of this dissertation wrote much of the code behind Encog and has extensive experience with the Encog framework.

The required equipment is currently available without restrictions. If additional hardware is needed, it can be acquired within a reasonable time to continue the research process. In the event of hardware failure, all equipment is readily available from multiple online sources for replacement within a week. All required software is currently available for the execution of this research, and the programming components have already been acquired. In the event of problems with the current software or catastrophic system failure of the system, the application development software is available for reacquisition from the original sources online.

Both the vendor and online community provide support for the programming environment in the event that there are issues with the software or with implementation of the various components. There is currently no anticipated need to perform interaction with end users or study participants because of the type of research project. There are no anticipated costs for hardware or software beyond Amazon AWS fees. If any Amazon AWS fees are incurred, they will be paid with the budget set aside to acquire additional and/or replacement hardware, software and processing fees. There will be no financial costs to Nova Southeastern University for this project.

Summary

This dissertation will leverage genetic programming to create an algorithm that can engineer features that might increase the accuracy of a deep neural network. Not all data sets will contain features that can be engineered into a better feature vector for the neural network. As a result, it is important to use a number of different data sets to evaluate the proposed algorithm. The effectiveness of the algorithm will be determined by evaluating the change in error between two neural networks—one has access to the algorithm's engineered features and the other does not.

Combining the knowledge from five planned experiments will create the proposed algorithm. The sixth experiment will perform a side-by-side benchmark between data sets augmented with features engineered by the algorithm and those that are not. The effectiveness of the algorithm can be measured by the degree to which the error decreases for the feature-engineered data set, when compared to an ordinary data set.

The five experiments will evaluate how to leverage different aspects of genetic programming for neural network feature engineering. Expressions that are beneficial to neural networks will be explored. Objective function design will be examined. Neural network feature ranking will be optimized for the quickest results. Ensembles will possibly detect data sets that could benefit the most from feature engineering. Information gained from all of these algorithms will guide the algorithm design.

After completing the project, the final dissertation report will be distilled and submitted as an academic paper to a journal or conference. At that point, the source code necessary to reproduce this research will be placed on the author's GitHub[1] repository. For reasons of confidentiality, the source code will not be publicly distributed prior to formal publication of the dissertation report.

[1] http://www.github.com/jeffheaton

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., . . . Devin, M. (2016). TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *arXiv preprint arXiv:1603.04467*.

Anscombe, F. J., & Tukey, J. W. (1963). The examination and analysis of residuals. *Technometrics, 5*(2), 141-160.

Arnold, K., Gosling, J., & Holmes, D. (1996). *The Java programming language* (Vol. 2): Addison-wesley Reading.

Bahnsen, A. C., Aouada, D., Stojanovic, A., & Ottersten, B. (2016). Feature Engineering Strategies for Credit Card Fraud Detection. *Expert Systems with Applications*.

Balkin, S. D., & Ord, J. K. (2000). Automatic neural network modeling for univariate time series. *International Journal of Forecasting, 16*(4), 509-515.

Banzhaf, W. (1993). *Genetic programming for pedestrians*. Paper presented at the Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93, University of Illinois at Urbana-Champaign.

Banzhaf, W., Francone, F. D., Keller, R. E., & Nordin, P. (1998). *Genetic programming: an introduction: on the automatic evolution of computer programs and its applications*: Morgan Kaufmann Publishers Inc.

Bastien, F., Lamblin, P., Pascanu, R., Bergstra, J., Goodfellow, I., Bergeron, A., . . . Bengio, Y. (2012). Theano: new features and speed improvements. *arXiv preprint arXiv:1211.5590*.

Bellman, R. (1957). *Dynamic Programming*. Princeton, NJ, USA: Princeton University Press.

Bengio, Y. (2009). Learning deep architectures for AI. *Foundations and trends in Machine Learning, 2*(1), 1-127.

Bengio, Y. (2013). Representation learning: a review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 35*(8), 1798-1828.

Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., . . . Bengio, Y. (2010). *Theano: a CPU and GPU math expression compiler.* Paper presented at the Proceedings of the Python for Scientific Computing Conference (SciPy).

Bertsekas, D. P. (1999). Nonlinear programming.

Bíró, I., Szabó, J., & Benczúr, A. A. (2008). *Latent Dirichlet allocation in web spam filtering.* Paper presented at the Proceedings of the 4th international workshop on Adversarial information retrieval on the web.

Bishop, C. M. (1995). *Neural networks for pattern recognition*: Oxford University Press.

Bizer, C., Heath, T., & Berners-Lee, T. (2009). Linked data-the story so far. *Semantic Services, Interoperability and Web Applications: Emerging Concepts*, 205-227.

Blei, D. M., Ng, A. Y., & Jordan, M. I. (2003). Latent Dirichlet allocation. *The Journal of Machine Learning Research, 3*, 993-1022.

Bottou, L. (2012). Stochastic gradient descent tricks *Neural Networks: Tricks of the Trade* (pp. 421-436): Springer.

Box, G. E. P., & Cox, D. R. (1964). An analysis of transformations. *Journal of the Royal Statistical Society. Series B (Methodological), 26*(2), pp. 211-252.

Breiman, L. (1996). Bagging predictors. *Machine learning, 24*(2), 123-140.

Breiman, L. (2001). Random forests. *Machine learning, 45*(1), 5-32.

Breiman, L., & Friedman, J. H. (1985). Estimating optimal transformations for multiple regression and correlation. *Journal of the American Statistical Association, 80*(391), 580-598.

Brosse, S., Lek, S., & Dauba, F. (1999). Predicting fish distribution in a mesotrophic lake by hydroacoustic survey and artificial neural networks. *Limnology and Oceanography, 44*(5), 1293-1303.

Brown, B. F. (1998). *Life and health insurance underwriting*: Life Office Management Association.

Brown, M., & Lowe, D. G. (2003). *Recognising panoramas.* Paper presented at the ICCV.

Chea, R., Grenouillet, G., & Lek, S. (2016). Evidence of Water Quality Degradation in Lower Mekong Basin Revealed by Self-Organizing Map. *PloS one, 11*(1).

Cheng, B., & Titterington, D. M. (1994). Neural networks: a review from a statistical perspective. *Statistical science*, 2-30.

Cheng, W., Kasneci, G., Graepel, T., Stern, D., & Herbrich, R. (2011). *Automated feature generation from structured knowledge.* Paper presented at the Proceedings of the 20th ACM International Conference on Information and Knowledge Management.

Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2015). Gated feedback recurrent neural networks. *arXiv preprint arXiv:1502.02367*.

Coates, A., Lee, H., & Ng, A. Y. (2011). *An analysis of single-layer networks in unsupervised feature learning*. Paper presented at the Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics.

Coates, A., & Ng, A. Y. (2012). Learning feature representations with k-means *Neural Networks: Tricks of the Trade* (pp. 561-580): Springer.

Colorni, A., Dorigo, M., & Maniezzo, V. (1991). *Distributed optimization by ant colonies.* Paper presented at the Proceedings of the First European Conference on Artificial Life.

Crepeau, R. L. (1995). *Genetic evolution of machine language software*. Paper presented at the Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications, Tahoe City, California, USA.

Cuayáhuitl, H. (2016). SimpleDS: A Simple Deep Reinforcement Learning Dialogue System. *arXiv preprint arXiv:1601.04574*.

Davis, J. J., & Foo, E. (2016). Automated feature engineering for HTTP tunnel detection. *Computers & Security, 59*, 166-185.

Dawkins, R. (1976). *The selfish gene*: Oxford university press.

De Boer, P.-T., Kroese, D. P., Mannor, S., & Rubinstein, R. Y. (2005). A tutorial on the cross-entropy method. *Annals of operations research, 134*(1), 19-67.

Deb, K. (2001). *Multi-objective optimization using evolutionary algorithms*: John Wiley & Sons, Inc.

Dietterich, T. G. (2000). Ensemble methods in machine learning *Multiple classifier systems* (pp. 1-15): Springer.

Diplock, G. (1998). Building new spatial interaction models by using genetic programming and a supercomputer. *Environment and Planning, 30*(10), 1893-1904.

Elman, J. L. (1990). Finding structure in time. *Cognitive science, 14*(2), 179-211.

Fisher, R. A. (1936). The use of multiple measurements in taxonomic problems. *Annals of eugenics, 7*(2), 179-188.

Freeman, M. F., & Tukey, J. W. (1950). Transformations related to the angular and the square root. *The Annals of Mathematical Statistics*, 607-611.

Fukushima, K. (1980). Neocognitron: a self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics, 36*(4), 193-202.

Garson, D. G. (1991). Interpreting neural network connection weights.

Glorot, X., & Bengio, Y. (2010). *Understanding the difficulty of training deep feedforward neural networks.* Paper presented at the International Conference on Artificial Intelligence and Atatistics.

Glorot, X., Bordes, A., & Bengio, Y. (2011). *Deep sparse rectifier neural networks.* Paper presented at the International Conference on Artificial Intelligence and Statistics.

Goh, A. (1995). Back-propagation neural networks for modeling complex systems. *Artificial Intelligence in Engineering, 9*(3), 143-151.

Graves, A., Wayne, G., & Danihelka, I. (2014). Neural turing machines. *arXiv preprint arXiv:1410.5401*.

Gruau, F. (1996). *On using syntactic constraints with genetic programming.* Paper presented at the Advances in Genetic Programming.

Guo, H., Jack, L. B., & Nandi, A. K. (2005). Feature generation using genetic programming with application to fault classification. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics, 35*(1), 89-99.

Guyon, I., Gunn, S., Nikravesh, M., & Zadeh, L. A. (2008). *Feature extraction: foundations and applications* (Vol. 207): Springer.

Gybenko, G. (1989). Approximation by superposition of sigmoidal functions. *Mathematics of Control, Signals and Systems, 2*(4), 303-314.

Heaton, J. (2015). Encog: library of interchangeable machine learning models for java and c#. *Journal of Machine Learning Research, 16*, 1243-1247.

Heaton, J. (2016). *An empirical analysis of feature engineering for predictive modeling*. Paper presented at the IEEE Southeastcon 2016, Norfolk, VA.

Hebb, D. O. (1949). The organization of behavior: New York: Wiley.

Hinton, G., Osindero, S., & Teh, Y.-W. (2006). A fast learning algorithm for deep belief nets. *Neural Computing, 18*(7), 1527-1554. doi:10.1162/neco.2006.18.7.1527

Hinton, G., & Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. *Science, 313*(5786), 504-507.

Hochreiter, S. (1991). Untersuchungen zu dynamischen neuronalen Netzen. *Diploma, Technische Universität München*.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural computation, 9*(8), 1735-1780.

Holland, J. H. (1975). *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. University of Michigan Press.

Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural networks, 4*(2), 251-257.

Ildefons, M. D. A., & Sugiyama, M. (2013). Winning the Kaggle Algorithmic Trading Challenge with the Composition of Many Models and Feature Engineering. *IEICE transactions on information and systems, 96*(3), 742-745.

Janikow, C. Z. (1996). A methodology for processing problem constraints in genetic programming. *Computers & Mathematics with Applications, 32*(8), 97-113.

Jones, E., Oliphant, T., Peterson, P., & al., e. (2001). SciPy: open source scientific tools for Python. Retrieved from http://www.scipy.org/

Jordan, M. I. (1997). Serial order: A parallel distributed processing approach. *Advances in psychology, 121*, 471-495.

Kalchbrenner, N., Danihelka, I., & Graves, A. (2015). Grid long short-term memory. *arXiv preprint arXiv:1507.01526*.

Kanter, J. M., & Veeramachaneni, K. (2015). *Deep feature synthesis: towards automating data science endeavors.* Paper presented at the IEEE International Conference on Data Science and Advanced Analytics (DSAA), 2015. 36678 2015. .

Kennedy, J. (2010). Particle swarm optimization *Encyclopedia of Machine Learning* (pp. 760-766): Springer.

Knuth, D. E. (1997). *The art of computer programming, volume 1 (3rd ed.): fundamental algorithms*: Addison Wesley Longman Publishing Co., Inc.

Koza, J. R. (1992). *Genetic programming: on the programming of computers by means of natural selection*: MIT Press.

Kuhn, M., & Johnson, K. (2013). Applied predictive modeling. New York, NY: Springer.

Le, Q. V. (2013). *Building high-level features using large scale unsupervised learning.* Paper presented at the Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on Acoustics.

Lloyd, J. R., Duvenaud, D., Grosse, R., Tenenbaum, J. B., & Ghahramani, Z. (2014). Automatic construction and natural-language description of nonparametric regression models. *arXiv preprint arXiv:1402.4304*.

Lowe, D. G. (1999). *Object recognition from local scale-invariant features.* Paper presented at the The Proceedings of the Seventh IEEE International Conference on Computer Vision, 1999.

Masters, T. (1993). *Practical neural network recipes in C++*: Morgan Kaufmann.

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics, 5*(4), 115-133.

McKinney, W. (2012). *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython*: O'Reilly Media, Inc.

Miller, J. F., & Harding, S. L. (2008). *Cartesian genetic programming*. Paper presented at the Proceedings of the 10th Annual Conference Companion on Genetic and Evolutionary Computation, Atlanta, GA, USA.

Miller, J. F., & Thomson, P. (2000). Cartesian genetic programming *Lecture Notes in Computer Science* (Vol. 1802, pp. 121-132): Springer.

Minsky, M. L., & Papert, S. A. (1969). Perceptrons. an introduction to computational geometry. *Science, 165*(3895).

Mosteller, F., & Tukey, J. W. (1977). Data analysis and regression: a second course in statistics. *Addison-Wesley Series in Behavioral Science: Quantitative Methods*.

Mozer, M. C. (1989). A focused back-propagation algorithm for temporal pattern recognition. *Complex systems, 3*(4), 349-381.

Nelder, J. A., & Mead, R. (1965). A simplex method for function minimization. *The computer journal, 7*(4), 308-313.

Neshatian, K. (2010). Feature Manipulation with Genetic Programming.

Nesterov, Y. (1983). *A method of solving a convex programming problem with convergence rate O (1/k2).* Paper presented at the Soviet Mathematics Doklady.

Newman, C. L. B. D. J., & Merz, C. J. (1998). UCI repository of machine learning databases.

Ng, A. Y. (2004). *Feature selection, L 1 vs. L 2 regularization, and rotational invariance.* Paper presented at the Proceedings of the twenty-first international conference on Machine learning.

Nguyen, D. H., & Widrow, B. (1990). Neural networks for self-learning control systems. *Control Systems Magazine, IEEE, 10*(3), 18-23.

Nordin, P. (1994). A compiling genetic programming system that directly manipulates the machine code. In K. E. Kinnear, Jr. (Ed.), *Advances in Genetic Programming* (pp. 311-331): MIT Press.

Nordin, P., Banzhaf, W., & Francone, F. D. (1999). Efficient evolution of machine code for CISC architectures using instruction blocks and homologous crossover. In L. Spector, W. B. Langdon, U.-M. O'Reilly, & P. J. Angeline (Eds.), *Advances in Genetic Programming* (Vol. 3, pp. 275--299). Cambridge, MA, USA: MIT Press.

Olshausen, B. A., & Field, D. J. (1996). Emergence of simple-cell receptive field properties by learning a sparse code for natural images. *Nature, 381*, 607--609.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., . . . Dubourg, V. (2011). Scikit-learn: Machine learning in Python. *The Journal of Machine Learning Research, 12*, 2825-2830.

Perkis, T. (1994). *Stack-Based Genetic Programming*. Paper presented at the International Conference on Evolutionary Computation.

Poli, R., Langdon, W. B., & McPhee, N. F. (2008). *A Field Guide to Genetic Programming*: Lulu Enterprises, UK Ltd.

Polyak, B. T. (1964). Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics, 4*(5), 1-17.

Prechelt, L. (1994). Proben1: A set of neural network benchmark problems and benchmarking rules.

Rajaraman, A., & Ullman, J. D. (2011). *Mining of massive datasets*: Cambridge University Press.

Robinson, A., & Fallside, F. (1987). *The utility driven dynamic error propagation network*: University of Cambridge Department of Engineering.

Rosenblatt, F. (1962). Principles of neurodynamics.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1985). *Learning internal representations by error propagation*. Retrieved from

Russell, S., & Norvig, P. (1995). Artificial intelligence: a modern approach.

Scott, S., & Matwin, S. (1999). *Feature engineering for text classification.* Paper presented at the ICML.

Smola, A., & Vapnik, V. (1997). Support vector regression machines. *Advances in neural information processing systems, 9*, 155-161.

Sobkowicz, A. (2016). Automatic Sentiment Analysis in Polish Language *Machine Intelligence and Big Data in Industry* (pp. 3-10): Springer.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research, 15*(1), 1929-1958.

Stanley, K. O., & Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary computation, 10*(2), 99-127.

Stigler, S. M. (1986). *The history of statistics: the measurement of uncertainty before 1900*: Belknap Press of Harvard University Press.

Sussman, G., Abelson, H., & Sussman, J. (1983). Structure and interpretation of computer programs: MIT Press, Cambridge, Mass.

Sutskever, I., Martens, J., Dahl, G., & Hinton, G. (2013). *On the importance of initialization and momentum in deep learning.* Paper presented at the Proceedings of the 30th International Conference on Machine Learning (ICML-13).

Teller, A. (1994). *Turing completeness in the language of genetic programming with indexed memory.* Paper presented at the Proceedings of the First IEEE Conference on Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence.

Timmerman, M. E. (2003). Principal component analysis (2nd Ed.). I. T. Jolliffe. *Journal of the American Statistical Association, 98*, 1082-1083.

Tukey, J. W., Laurner, J., & Siegel, A. (1982). The use of smelting in guiding re-expression *Modern Data Analysis* (pp. 83-102): Academic Press New York.

Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Journal of Math, 58*(345-363), 5.

Van der Maaten, L., & Hinton, G. (2008). Visualizing data using t-SNE. *Journal of Machine Learning Research, 9*(2579-2605), 85.

Van Rossum, G. (1995). Python tutorial, May 1995. *CWI Report CS-R9526*.

Wang, C., & Blei, D. M. (2011). *Collaborative topic modeling for recommending scientific articles.* Paper presented at the Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.

Wang, M., Li, L., Yu, C., Yan, A., Zhao, Z., Zhang, G., . . . Gasteiger, J. (2016). Classification of Mixtures of Chinese Herbal Medicines Based on a Self-Organizing Map (SOM). *Molecular Informatics*.

Werbos, P. (1974). Beyond regression: new tools for prediction and analysis in the behavioral sciences.

Werbos, P. (1988). Generalization of backpropagation with application to a recurrent gas market model. *Neural networks, 1*(4), 339-356.

White, D. R., Mcdermott, J., Castelli, M., Manzoni, L., Goldman, B. W., Kronberger, G., . . . Luke, S. (2013). Better GP benchmarks: community survey results and proposals. *Genetic Programming and Evolvable Machines, 14*(1), 3-29.

Worm, T., & Chiu, K. (2013). *Prioritized grammar enumeration: symbolic regression by dynamic programming*. Paper presented at the Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation, Amsterdam, The Netherlands.

Yu, D., Eversole, A., Seltzer, M., Yao, K., Huang, Z., Guenter, B., . . . Wang, H. (2014). *An introduction to computational networks and the computational network toolkit*. Retrieved from

Yu, H.-F., Lo, H.-Y., Hsieh, H.-P., Lou, J.-K., McKenzie, T. G., Chou, J.-W., . . . Chen-Wei, H. (2011). *Feature engineering and classifier ensemble for KDD cup 2010.* Paper presented at the JMLR: Workshop and Confrence Proceedings 1: 1-16.

Zhang, W., Huan, R., & Jiang, Q. (2016). Application of Feature Engineering for Phishing Detection. *IEICE transactions on information and systems, 99*(4), 1062-1070.

Ziehe, A., Kawanabe, M., Harmeling, S., & Müller, K.-R. (2001). *Separation of post-nonlinear mixtures using ACE and temporal decorrelation.* Paper presented at the Proceedings of the International Workshop on Independent Component Analysis and Blind Signal Separation (ICA2001).