Automated Feature Engineering for Deep Neural Networks with Genetic Programming

by

Jeff Heaton

An idea paper 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

April 2016

Abstract

Feature engineering is a process that augments the feature vector of a predictive model with calculated values that are designed to enhance the model's performance. Models such as neural networks, support vector machines and tree/forest-based algorithms have all been shown to sometimes benefit from feature engineering. Engineered features are created by functions that combine one or more of the original features presented to the model.

The choice of the exact structure of an engineered feature is dependent on the type of machine learning model in use. Previous research shows that tree based models, such as random forests or gradient boosted machines, benefit from a different set of engineered features than dot product based models, such as neural networks and multiple regression. The proposed research seeks to use genetic programming to automatically engineer features that will benefit deep neural networks. Engineered features generated by the proposed research will include both transformations of single original features, as well as functions that involve several original features.

Introduction

This paper presents proposed research for an algorithm that will automatically engineer features that will benefit deep neural networks for certain types of predictive problems. The proposed research builds upon, but does not duplicate, prior published research by the author. In 2008 the author introduced the Encog Machine Learning Framework that includes advanced neural network and genetic programming algorithms (Heaton, 2015). The Encog genetic programming algorithm introduced an innovative method that allows dynamic constant nodes, rather than the static constant pool typical used by tree based genetic programming.

The author of this dissertation also performed research that demonstrated the types of manually engineered features most conducive to deep neural networks (Heaton, 2016). The proposed research builds upon this prior research by leveraging the Encog genetic programming algorithm to be used in conjunction with 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).

This paper begins with an introduction of both neural networks and feature engineering. The problem statement is defined and a clear dissertation goal is given. Building upon this goal, a justification is given for the relevance of this research, along with a discussion of the barriers and issues previously encountered. A brief review of literature is provided to show how this research continues previous research in deep learning. The approach that will be used to achieve the dissertation goal is given, along with the necessary resources and planned schedule.

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. These inputs are called features and the complete set of inputs is called a feature vector. Many different types of data, such as pixel grids for computer vision or named attributes describing business data can be mapped to the neural network's inputs (B. F. Brown, 1998).

Most business applications of neural networks must map input neurons to columns in a database, this input is used to make a prediction. For example, an insurance company might use the columns: 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). Regression neural networks will output a real number, such as the maximum face amount to issue the applicant. Classification neural networks will output a class that the input belongs to. *Figure 1* shows both of these neural networks.

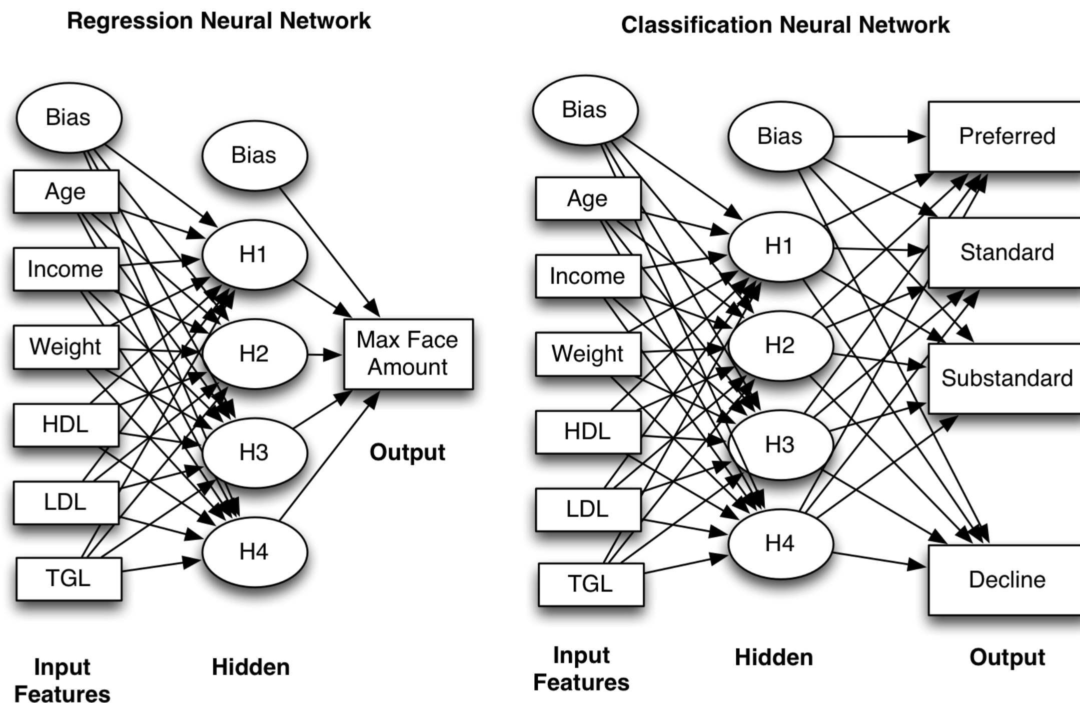

Figure 1. Regression and classification network (original features)

The left neural network performs a regression and uses the original 6 input features to determine the maximum face amount to issue an applicant. The right neural network performs a classification and uses the same original 6 input features to determine the underwriting class to place the insured into. The weights (shown as arrows) determine what the final output will be. A backpropagation algorithm uses many sets of inputs with known output to determine the weights. The neural network learns from existing data to predict future data. The above networks have a single hidden layer. However, additional layers can be added. Deep neural networks might have between 5 and 10 hidden layers between the input and output layers. A bias neuron is commonly added to every layer except the output layer. The bias neuron always outputs a consistent value, usually 1.0.

Bias neurons greatly enhance the neural network's learning ability (B. Cheng & Titterington, 1994).

The output of a neural network is determined by the output neurons. Processing the value of each neuron, starting from the input layer, allows the output neurons to be calculated. *Equation 1* is used to calculate each neuron in a neural network.

$$f(x, w, b) = \phi\left(\sum_i (w_i x_i) + b\right)$$

Equation 1. Neuron calculation

The function phi (Φ) represents the transfer function, and 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 referred to as being part of a larger class of machine learning algorithms that are dot product based.

Feature engineering simply adds additional 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 to understand the interaction of several fields in the original dataset. For example, human insurance underwriters benefit from combining height and weight to calculate BMI. Likewise, human insurance underwriters often use a ratio of the HDL, TGL and LDL cholesterol levels. These calculations allow a single number to be compared to evaluate the health of the applicant. These

calculations might also be useful to the neural network. If the BMI and HDL/LDL ratio were engineered as features, the classification network would look like *Figure 2*.

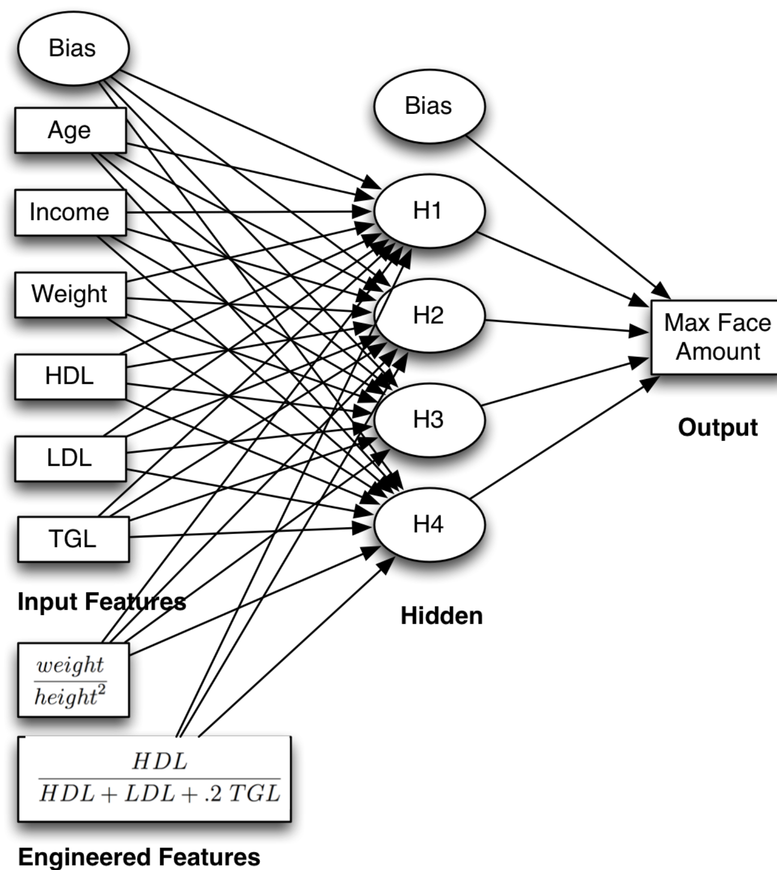

Figure 2. Neural network engineered features

In *Figure 2* the BMI and HDL/LDL ratio values are added to the feature vector along with the original input features. The entire feature vector is provided to the neural network. These additional features might help the neural network to calculate the maximum face amount. These two features could also be used for classification. 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 relatively easy, as these are well known

calculations. Similar calculations might also benefit other datasets. Feature engineering often involves combining original features with ratios, summations, differences and power functions. Engineered features, such as BMI, that involve multiple original features, are usually derived manually using intuition about the dataset.

Problem Statement

There is currently no automated means of engineering features for a deep neural network that are a combination of multiple values from the original feature vector. Prior feature engineering work has focused primarily upon transformation of a single feature or upon models other than deep learning (Box & Cox, 1964; Breiman & Friedman, 1985; Freeman & Tukey, 1950). Feature engineering research focused on 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 is heavily dependent 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 (Geoffrey E. Hinton, Osindero, & Teh, 2006) can benefit from feature engineering. Most research into feature engineering in the deep learning space 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 (G E Hinton & Salakhutdinov, 2006) such

as Principal Component Analysis (PCA) (Timmerman, 2003) and auto-encoders (Olshausen & Field, 1996).

Dissertation Goal

The goal of this research is to use genetic programming to analyze a dataset and automatically create engineered features that will benefit a deep neural network. These engineered features will consist of mathematical transformations of one or more of the existing features from the dataset. In completing this research, the successful outcome will be an algorithm that derives successfully engineered features for an array of synthetic and real datasets. A successfully engineered feature is a feature that decreases the neural network's prediction loss function when that feature is added to the input feature vector of the neural network.

The proposed research focuses on datasets consisting of named features, as opposed to datasets that contains large amounts of pixels or audio sampling data. A named feature is considered a value such as income, height, weight or gender; named features are not considered to be the color of a pixel at a particular coordinate or an audio sample at a time-slice. For many predictive modeling applications, such as fraud monitoring, sales forecasting, and intrusion detection, the features consist of these named values. Real-world datasets, used for predictive modeling, can consist of a handful or several hundred such features. The proposed algorithm would be ineffective for computer vision and hearing applications.

Relevance and Significance

Since the introduction of multiple linear regression in the late 1800's statisticians have been employing creative means to transform model input to enhance model output (Anscombe & Tukey, 1963; Stigler, 1986). These transformations usually applied a single mathematical formula to an individual feature. For example, one feature might have a logarithm applied; whereas, another feature might be raised to the third power. Such transformations can significantly increase the performance of a model (Kuhn & Johnson, 2013).

Considerable research has been focused on automated derivation of such transformations. Freeman and Tukey (1950) reported on a number of transformations useful for linear regression. Box and Cox (1964) conducted the seminal work on automatic feature transformation and used a stochastic ad hoc algorithm to automatically recommend transformations that might improve upon the results of linear regression. This became known as the Box Cox transformation. While the Box Cox transformation was capable of obtaining favorable transformations for linear regression, due to its stochastic nature it often did not converge to the best transformation for an individual feature. Numerous other transformations were created that used 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. While all of the aforementioned algorithms were

designed for linear regression, they are often used to assist tree models, neural networks and other models. Because ACE as 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 to the features independently. Features that transform several original input features collectively can produce favorable results (Yu et al., 2011).

Kaggle and ACM's KDD Cup have seen feature engineering play an important role in several winning submissions. Yu et al. (2011) reported on the successful application of feature engineering to the KDD Cup 2010 competition. Ildefons and Sugiyama (2013) were able to win 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

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

The output from a linear regression is simply a multi-term first-degree equation. Because of its simplicity several mathematical techniques can be used to guarantee optimal transformations. Deep neural networks have a much more complex structure

than linear regression. It will be a different challenge to engineer features for a deep neural network than it is to engineer them for linear regression.

Previous automated feature engineering algorithms focused on the transformation of a single original input feature. They do not combine multiple features to generate engineered features such as BMI (that combines height and weight) or the HDL/LDL ratio (that combines HDL, LDL and TGL). Searching for engineered features that involve multiple original features will greatly increase the search space for engineered features. This expanded search space will face the curse of dimensionality (Bellman, 1957) and require novel solutions to limit and prioritize the search space.

This problem is also difficult because not all datasets benefit from feature engineering. Because of this it will be necessary to make use of a number of different datasets from the UCI Machine Learning Repository (Newman & Merz, 1998). It will also be necessary to generate several synthetic datasets that are designed to benefit from certain types of features. It will take considerable experimentation to try all selected real and synthetic datasets against the proposed solution.

Brief Review of Literature

The research proposed by this paper focuses primarily upon feature engineering and how to apply it to deep neural networks. Evolutionary programming will be used to provide an automated system to recommend new 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 interest in all of these areas. The following sections review current literature in these areas as it pertains to the proposed research.

Feature Engineering

Feature engineering grew out of the need to transform linear regression inputs that are not normally distributed (Freeman & Tukey, 1950). Such transformation is necessary because linear regression assumes normally distributed input. The seminal work on automated feature engineering was introduced by Box and Cox (1964) and showed a method for determining which of several power functions might be a useful feature transformation for the inputs to linear regression. Power transformations simply apply exponents to the input features of a machine-learning model. Other mathematical functions may also be used for transformation with logarithms being one popular choice. Linear regression is not the only machine-learning model that benefits from feature engineering transformations. These simple transformations simply modify the individual features independently of each other.

The method proposed by 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 ACE algorithm that could guarantee optimal transformations for linear regression. The ACE algorithm finds a set of optimal transformations for each of the predictor features, as well as the outcome to be used with linear regression. Though the resulting transformations were originally intended for linear regression, they have been used 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 of 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 many functions, such as log and power functions. Brosse, Lek, and Dauba (1999) made use of 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 to solve one of the biggest problems in computer vision—image scaling. A machine learning model that learns to recognize digits might not recognize these same digits if their size is doubled. SIFT preprocesses the data and provides them in a form where images of multiple scales produce features that are much the same. These types of features can be successfully 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) made use of feature engineering to enhance the performance of rules learning for text classification. These transformations allow characteristics about the text, such as structure and frequency to be generalized to just a few features. Representing textual data to a machine learning model produces a considerable number of dimensions. Feature engineering to reduce these dimensions is useful for text classification.

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 several document classification tasks, such as spam filtering (Bíró, Szabó, & Benczúr, 2008) and article recommendation (Wang & Blei, 2011).

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

Feature engineering has proven successful in data science competitions, such as Kaggle and KDD Cup. One early use of feature engineering for a competition was the KDD Cup 2010 competition, that was won by a team that successfully applied feature engineering and an ensemble of machine learning models (Yu et al., 2011). Other features, such as histograms of oriented gradients were useful in this competition. W. Cheng, Kasneci, Graepel, Stern, and Herbrich (2011) developed an automated feature generation algorithm for data organized with domain-specific knowledge. Later, 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 by hand.

Natural language processing (NLP) has benefited greatly from feature engineering. 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 how frequently a word shows up in a document to how often it shows up in the whole corpus of documents (Rajaraman & Ullman, 2011). TF-IDF has proven popular for text mining, text classification and NLP.

Machine learning algorithms themselves have been used to perform automated feature learning. These algorithms are often unsupervised, as they are not provided an expected outcome. Coates et al. (2011) used an unsupervised single layer ANN for feature engineering. Dimension reduction algorithms, such as principal component analysis (PCA) (Timmerman, 2003) and t-distributed stochastic neighbor embedding (T-SNE) (Van der Maaten & Hinton, 2008) have also proven successful for automated feature engineering in some cases. Other unsupervised machine learning algorithms have also proven successful at feature engineering, Coates and Ng (2012) made use of 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 several classes of deep learning problem, such as: 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) made use of feature engineering to create the *Automatic Statistician* project. This system automatically models regression problems and produces reports readable by humans. This system can automatically determine the types of transformations that might benefit individual features.

Kanter and Veeramachaneni (2015) invented a technique called deep feature synthesis, that can be used to automatically transform relational database tables into the feature vector needed by the typical machine learning model. The deep feature synthesis makes use of SQL-like transformations, such as MIN, MAX, and COUNT to summarize one-to-many and many-to-many relationships. The deep feature synthesis algorithm was able to outperform the majority of competitors in three data science competitions.

Neural Networks

Neural networks are a loosely biologically-inspired class of algorithms that were introduced by McCulloch and Pitts (1943) as networks made up of MP-Units. This seminal algorithm 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. There has been considerable research into automated means of selecting weights for a neural network that 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 between the neurons necessary for that action increase. This process became Hebb's rule, and 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 used by the 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; and returns 0 otherwise. Severe limitations in the perceptron were described by Minsky and Papert (1969) in their monograph. They demonstrated that perceptrons were incapable of learning non-linearly separable problems, such as the exclusive or (XOR) operator.

Research continued into a means of automatically determining the weights of a neural network. What would become the backpropagation algorithm was introduced by Werbos (1974) in his PhD thesis. Backpropagation is the seminal algorithm that forms the basis for automatically determining neural network weights from data. Backpropagation is based on gradient descent, which is a supervised training algorithm where a series of observations are provided that specify the expected outcome for input vectors to the neural network. This algorithm adjusts the weights by calculating the partial derivatives of each weight for the loss function, with respect to the other weights. These partial derivatives are called the gradients, and provide an indication of how to change these weights to minimize the loss function. The difference between the neural

networks' current and expected outputs is the loss function for a neural network. The loss function of a neural network is often called its error function.

Gradient descent was first applied to neural networks by Rumelhart, Hinton, and Williams (1985). Their algorithm is called the backward propagation of errors, or backpropagation. The gradient of each weight is calculated and used to determine a change that should occur in the weight for the current training epoch. The gradient of each weight is essentially the partial derivative of the loss function for that weight with all other weights held constant. Backpropagation is essentially the application of gradient descent to neural network training.

Backpropagation was initially ineffective at training neural networks with significantly more than two hidden layers. It was also not initially understood if neural networks actually benefited from many layers. Gybenko (1989) formulated the universal approximation theorem, proving that a single hidden-layer neural network could approximate any function. This research was built upon by Hornik (1991), who proved that it was not the specific choice of the transfer function, but rather the multilayer feedforward architecture itself which gives neural networks the potential of being universal approximators. The universal approximation theorem essentially implies that because a single hidden-layer neural network can theoretically learn any problem, additional hidden layers are unnecessary.

The inability to learn large numbers of hidden layers was not the only barrier to widespread neural network adoption. One considerable obstacle to the adoption of neural networks is the large number of hyper-parameters that a neural network contains. A neural network practitioner must decide how many layers the network must have and

how many hidden neurons will be contained by each of these hidden layers. Automatic determination of an optimal structure for neural networks is an ongoing area of research. Stanley and Miikkulainen (2002) invented the NEAT neural network that makes use of a genetic algorithm to optimize the neural network structure. The genetic algorithm searches for the optimal neural network structure and weight values to minimize the loss function.

Research continued on neural networks through the early 2000's. Neural networks were applied to a variety of tasks, such as time series prediction (Balkin & Ord, 2000). Most training algorithms for neural networks were not effective at training more than two layers. The ineffectiveness of training deep neural networks, combined with the universal approximation theorem's assertion that only one hidden layer is actually needed led to a stagnation of research into neural networks deeper than a single hidden layer.

Deep Learning

While it is theoretically possible for a single-hidden layer neural network to learn any problem (Hornik, 1991), this will not necessarily happen in practice. The additional hidden layers can allow the neural networks to learn hierarchies of features. This can simplify the search space and allow an optimal set of weights to be found with less training. Unfortunately, there was no method to train these networks until a series of innovations would make this training possible, and give rise to what is now referred to as deep learning. The output of a deep neural network is still calculated by passing the output of a dot product to a transfer function. The innovations, referred to as deep learning, introduce new transfer functions and training methods. The seminal calculation, given by *Equation 1*, introduced by McCulloch and Pitts (1943) remains intact.

The first successful application of a deep neural network was by Geoffrey E. Hinton et al. (2006). They created a learning algorithm that could train deep variants of the type of belief networks introduced by Fukushima (1980). This discovery renewed interest in deep neural networks. Several additional technologies, such as stochastic gradient descent, rectified linear units, and Nesterov momentum, have been introduced that have made training of deep neural networks more efficient. Taken together, these technologies are referred to as deep learning.

Prior to deep learning, most neural networks made use of a simple quadratic error function on the output layer (Bishop, 1995). The cross entropy error function, introduced by De Boer, Kroese, Mannor, and Rubinstein (2005) often achieves better results than the simple quadratic. The cross entropy error function makes use of the logarithm function and provides a more granular means of error representation than the quadratic error function.

Neural networks must start with random weights. Often these neural networks are initialized with random values within a range. One common choice for this range is the real numbers between -1 and 1. Simple range initialization can occasionally produce a set of weights that are difficult to train. There has been considerable research interest into weight initialization algorithms that provide a good set of starting weights for backpropagation (Nguyen & Widrow, 1990). Currently, the most popular weight initialization method is the one introduced by Glorot and Bengio (2010) that is called the Xaiver weight initialization algorithm. Xaiver weight initialization has since become a popular initialization algorithm for deep neural networks. It is rare that Xaiver weight

initialization will produce a set of initial random weights that are impossible to be trained by backpropagation.

Backpropagation relies on the derivatives of the transfer functions to propagate error corrections from the output neurons back through the weights of a neural network. Neural networks with many layers will often experience a problem where the gradients become zero with certain transfer functions. This is referred to as the vanishing gradient problem, and was first described by Hochreiter (1991) in his thesis. Prior to 2011, most neural network hidden layers made use of a sigmoidal transfer function, such as the hyperbolic tangent or the logistic transfer function. Both of these functions saturate to zero as x approaches either positive or negative infinity—this causes these transfer functions to exhibit the vanishing gradient problem. To alleviate this problem, Glorot, Bordes, and Bengio (2011) introduced the rectified linear unit (ReLU) transfer function.

The ReLU transfer function usually achieves much better training results for deep neural networks than the sigmoidal transfer functions more traditionally used. 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 use the ReLU transfer function. For their output layer, most deep neural networks make use of linear transfer function for regression, and a softmax transfer function for classification. No transfer function is needed for the input layer.

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 even the outliers in the dataset—such a neural network is learning to memorize, not generalize (Russell & Norvig, 1995). Algorithms designed to combat overfitting are

called regularization algorithms. Dropout was introduced by Srivastava, Hinton, Krizhevsky, Sutskever, and Salakhutdinov (2014) as a simple regularization technique for deep neural networks.

Another significant innovation that benefits deep learning is Nesterov momentum (Sutskever, Martens, Dahl, & Hinton, 2013). Momentum has been an important component of backpropagation training for some time. Simple momentum is a regularization technique that was originally introduced for gradient ascent by Polyak (1964). Momentum backpropagation simply adds a portion of the previous iteration's weight change to the current iteration's weight change. This effectively gives the weight changes the necessary momentum to continue through local minima and continue the descent to better loss function result levels. Nesterov momentum (Nesterov, 1983) further enhances the momentum calculation and increases the effectiveness of stochastic gradient descent (SGD). The SGD algorithm works by selecting mini-batches at each iteration for training that are randomly sampled from the training data. Nesterov momentum decreases the likelihood of a particularly bad mini-batch from changing the weights into too poor of a state.

Evolutionary Programming

Evolutionary algorithms were introduced by Holland (1975). Later Deb (2001) extended this work to introduce genetic algorithms as, "a generic population-based metaheuristic technique used to find solutions to many real-world search and optimization problems." These algorithms are inspired by Darwinian evolution. 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 essentially a search with the classic balance between exploitation and exploration. Mutation and crossover provide the genetic algorithm with the ability to explore and exploit the search space. The mutation genetic operator provides exploration by introducing randomness to the population, whereas the crossover genetic operator exploits by creating new members containing traits from the best members of the population (Holland, 1975).

One of the most commonly used evolutionary algorithms are the type of genetic algorithm introduced by Holland (1975). This algorithm represents potential solutions as fixed-length vectors that represent some abstraction of the solution to a problem. This vector might represent the weights of a neural network, coefficients of an equation, or any other fixed-length vector that must be optimized against an objective function. Mutation is accomplished by perturbing the elements of a vector in some way. Crossover is accomplished by splicing together the vectors of two or more parent vectors.

The population is evaluated using an objective function. 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 that is to be minimized. Some evolutionary algorithms also allow the objective function to be maximized. The choice between minimization and maximization is dependent on the domain of the problem. Though the loss function and objective function both accomplish similar goals, it is convention 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 of classic genetic algorithms. While a genetic algorithm that is evolving the weights of a neural network might produce better weights, it will never produce an improvement to the underlying neural network algorithm. To evolve better algorithms the computer programs themselves must become the genomes that will be evolved (Poli, Langdon, & McPhee, 2008). Genetic programming was created to evolve the computer programs themselves and overcome the limiting fixed-length vector of classic genetic algorithms.

Genetic programming evolves representations of actual computer programs to achieve an optimal score to a loss function. This active area of research was popularized by Koza (1992) as a means of automatically generating programs to solve specific problems. The majority of Koza's research represents the genetic programs as trees. Though the vast majority of genetic programming research has revolved around the tree representation, there is substantial research into other representations of the genetic programs (Wolfgang Banzhaf, Francone, Keller, & Nordin, 1998).

A tree-based genetic program is implemented as a directed acyclic graph (DAG). The tree is made up of connected nodes. The tree starts with a parentless root node, the root node connects to other nodes that, in turn, point to more nodes. The tree is made up of interior nodes that have at least one child. There are also leaf nodes that have no children. Ultimately the tree reaches all of the terminal nodes (Wolfgang Banzhaf et al., 1998). The terminal nodes represent variables and constants. The interior nodes make up the operators that use these variables and constants. *Equation 2* could be represented as a tree for genetic programming.

$$\frac{x}{2} - 1 + 2\cos(y)$$

Equation 2. Equation for genetic programming

It is common in computer science to represent such equations as trees. Early programming languages, such as Lisp and Scheme made extensive use of similar tree representations, called S-Expressions (Sussman, Abelson, & Sussman, 1983). *Equation 2* represented as a tree is given by *Figure 3*.

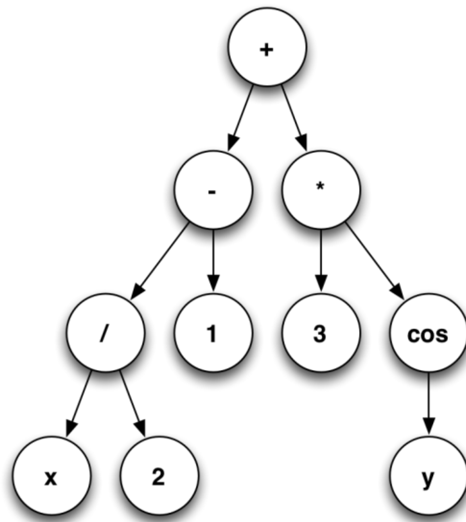

Figure 3. Equation tree for genetic programming

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 such trees can be Turing complete (Turing, 1936), which means they can theoretically compute anything. Additionally, nodes can be created that allow the values of variables to be changed. Trees are not the only representation of genetic programs. Much of the research into genetic programming has been to determine the best way to represent the genetic programs (Poli et al., 2008).

Modern computers represent computer programs as a series of linear instructions (Knuth, 1997), not as trees. Though these programs can be written as trees, it is often not practical, as the linear nature of programming will often create deep 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 linear 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) all went even further and evolved bit patterns that represented the actual CPU machine language instruction codes. The code created by the linear genetic programming systems closely resembles pseudo-code. This makes linear genetic programs easier for a human programmer to interpret 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 equations and other computational structures. These integer-based grids are represented as fixed-length vectors for the crossover and mutation genetic operators.

A grid, of a fixed size, is used to encode a CGP. In many ways, the encoded CGP resembles the sort of breadboard normally used in electronics. This grid has rows that are equal to the size of the feature vector used with a CGP algorithm. The number of columns is variable, but defines the complexity of the equations, or programs, that can be represented. In this way the number of columns is analogous to the number of hidden

neurons in an artificial neural network. A simple CGP grid, with two input variables (*x* and *y*) and one output is shown in *Figure 4*.

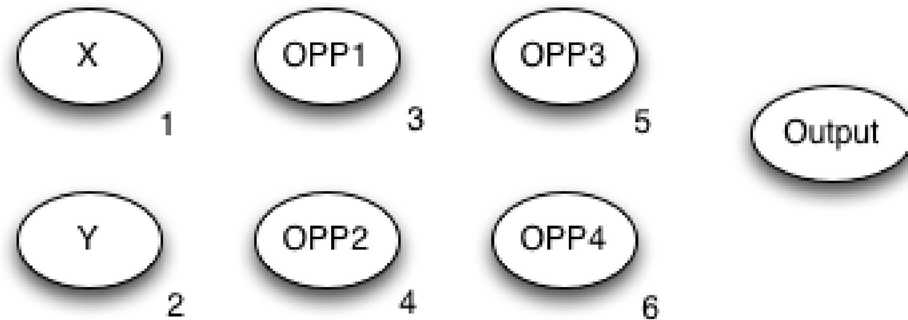

Figure 4. Cartesian genetic program (CGP)

A genome, using the above matrix representation, might be encoded as:

```
0 1 2 0 1 1 2 3 2 0 1 1 5
```

Grouping the genes in this genome, by operator, would result in:

```
[0 1 2] [0 1 1] [2 3 2] [0 1 1] [5]
```

The first four bracketed numbers encode OPP1 through OPP4. The first number in each of these three-value tuples provides the operator. For this example, the following operators are used:

- 0 represents addition.
- 1 represents subtraction.
- 2 represents multiplication.
- 3 represents division.

The first tuple requests that connections are added to indexes 1 and 2, yielding $x+y$. The final tuple, which contains only 5, specifies that index 5 is the output. In this case

the output is the output of index 5, which is OPP3. If every connection in the above genome were plotted, this would result in *Figure 5*.

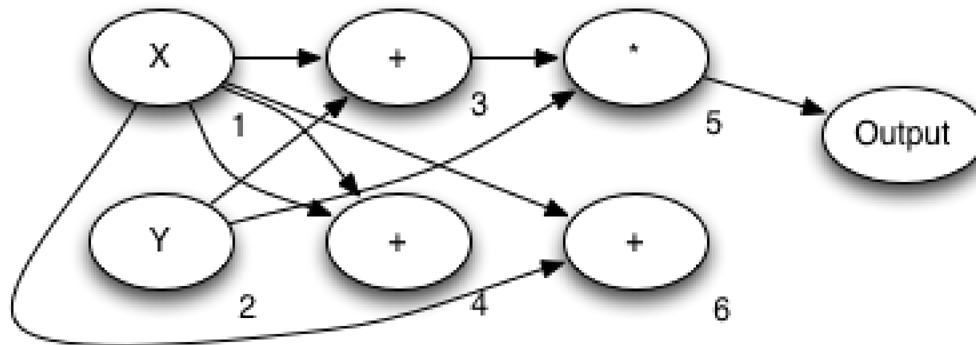

Figure 5. Cartesian genetic program (CGP) links

The above genome results in *Equation 3*.

$$y(y + x)$$

Equation 3. Equation for Cartesian genetic program

Of course, the above equation can be algebraically simplified; however, genetic programming typically does not perform this simplification. Another important concept is that much of the genome is not used. Only the parts of the genome that contain connections to the output are actually used. The genes that do not contribute to the output are called non-coding genes, which is a reference to biological genes that do not code protein. Additionally, inputs can also be non-coding. If the evolutionary process does not find one of the input features to be valuable, that feature will not be coded into the final program. This allows CGP to perform feature selection for dimensionality reduction (Mladenić, 2006) as an inherent part of its process. Feature selection must be performed as a separate step for many other machine learning algorithms.

Genetic programming has been applied to intrusion detection systems (IDS). Mabu, Chen, Nannan, Shimada, and Hirasawa (2011) made use of tree-based genetic programming to automatically generate rules for IDS. Using the KDD99 dataset and the DARPA98 databases from MIT Lincoln Laboratory, the researchers were able to evolve an effective set of IDS rules. The researchers noted that the tree structure of genetic programming was more effective than the fixed-length vector typically used in traditional genetic algorithm systems.

Many genetic programs operate on a single type of data—often floating point. However, computer programs often make use of many different data types. It would be useful to offer this same flexibility to genetic programs. Data types such as *integer*, *string* and *struct* could be useful to genetic programs. This complicates genetic programming. Not every operator can accept all types. For example, the "-" operator can easily be applied to integers and floating point numbers, but is undefined for strings or Booleans. To overcome this limitation, Worm and Chiu (2013) made use of a system of grammar rules. The grammar rules document what operators are depended on what type of data. These rules restrict the crossover and mutation operators and ensure that new programs are valid.

The research for this dissertation proposal covers a number of areas of literature, including: neural networks, deep learning, and genetic programming. Recent research in each of these areas was covered in the preceding sections. The proposed research seeks to use the latest advances in all of these areas to produce an algorithm to generate features for deep neural networks.

Approach

The proposed research is to create an algorithm capable of the automated creation of engineered features that would benefit a deep neural network. Unlike much of the previous feature engineering research, these engineered features are allowed to draw upon multiple original features. This problem can essentially be thought of as an infinite search over all combinations of the original feature set. The algorithm will see which combinations enhance the learning of the deep neural network. Obviously, such an approach is not possible. However, rather than try every possible combination of original features, a metaheuristic search algorithm would be used. Because the search space is potential equations, genetic programming is a natural choice of 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 also be used in conjunction with the genetic programming. However, the primary research direction for this proposed research should be genetic programming.

Narrowing the Search Domain

It is possible to place search constraints upon a genetic programming algorithm (Gruau, 1996; Janikow, 1996). Such constraints are sometimes implemented as an additional objective, producing a multi-objective genetic program. The primary objective remains achieving a favorable score from the deep learning loss function; however, the secondary objective severely penalizes equations of the type that should not be explored. In prior research the author determined that deep neural networks are not benefited by certain classes of function (Heaton, 2016), such as simple power functions, ratios,

differences and counts. This research would be expanded upon to determine additional useful equation structures for deep learning. Only features that follow these structures would be engineered.

Genetic programming requires a palette of operators to construct equations from. The selection of such operators is critical to a good solution. For example, some problems may benefit from adding the trigonometry functions. However, for other problems the trigonometry functions might be superfluous. Additionally, sometimes problem specific functions are added to the operator set for genetic programming. For this research the operator palette will be kept small; however, some functions that are specific to feature engineering will be added. The following palette operators are initially 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 reduce the dimensions of the feature vector to 3 and calculates the distance between the feature vector for the current dataset item and the centroid of each class in a classification problem. If the dataset is not used for

classification, then this operator will not be available. In addition to the above operators, others 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*. If this equation is examined, it can be seen that neural networks inherently have the ability to multiply and sum. Because of this, neural networks do not tend to benefit as much from engineered features involving simple multiplication and addition (Heaton, 2016). Because of this, it might not be worth the time to discover an engineered feature as complex as *Equation 4*.

$$f_e = \frac{3f_1f_2}{2f_3}$$

Equation 4. First engineered feature

In the above equation, an engineered feature (f_e) was calculated making use of 3 of the original features. Because neural networks can perform their own multiplication, it might be possible to simplify *Equation 4* to *Equation 5*.

$$f_e = \frac{f_1f_2}{f_3}$$

Equation 5. Second engineered feature

The feature engineered by *Equation 4* is not mathematically equivalent to *Equation 5*; however, *Equation 5* might be sufficient. This is because the neural network would have the ability to multiply 3/2 by the engineered feature. Because the proposed research is targeted at deep learning, it is important to not engineer parts of the feature that the neural network can easily learn during training.

Creating an Efficient Objective Function

The genetic programming algorithm used in this research will need an objective function to guide the genetic selection operator. This research 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 usefulness of engineered features. To do this, a control neural network will first be trained with only the original features present. Training of the control neural network will only need to be performed one time, at the start of training. To evaluate a potential engineered feature, it will be added to the feature vector and a new neural network trained. 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 zero do not improve the neural network's predictive power; whereas, scores below zero improved the neural network. The genetic programming algorithm will seek to decrease these scores.

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 number of training iterations. This will decrease 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, but rather to gain some indication of how effective the engineered features are. Other novel techniques will be evaluated to produce an objective function with acceptable performance.

The second objective will be to avoid the equations that are known to not improve a deep neural network's effectiveness. The author performed preliminary research and determined several feature classes that were not particularly helpful to deep neural networks (Heaton, 2016). From the author's prior research it was determined that counts, differences, logs, power functions, rational polynomials, and radicals were not particularly effective engineered features for deep learning. Similarly, our 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.

Generating Candidate Datasets

Not all datasets will benefit from engineered features. This is true if the underlying data do not contain relationships that can be exposed by feature engineering. Because of this, it will be necessary to create datasets that are designed to benefit from features that are known to be useful to deep neural networks. The feature engineering algorithm proposed by this research will be tested to see if it is capable of finding the engineered features that are known to help these generated datasets.

Datasets will be generated 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 the author devised a simple means to learn the types of features that benefit a particular machine learning model type, such as a deep neural network. The author simply generated

training sets where 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 is similar to the common neural network example of teaching a neural network 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 useful engineered feature.

*Figure **6***, from Heaton (2016), shows the effectiveness of a deep neural network learning to synthesize several types of engineered feature. The x-axis lists the types of engineered feature and the y-axis shows the root mean square error (RMSE) loss function result achieved while trying to synthesize the feature. Errors are capped at 0.05, if the RMSE error is above that level then the model is assumed to have failed.

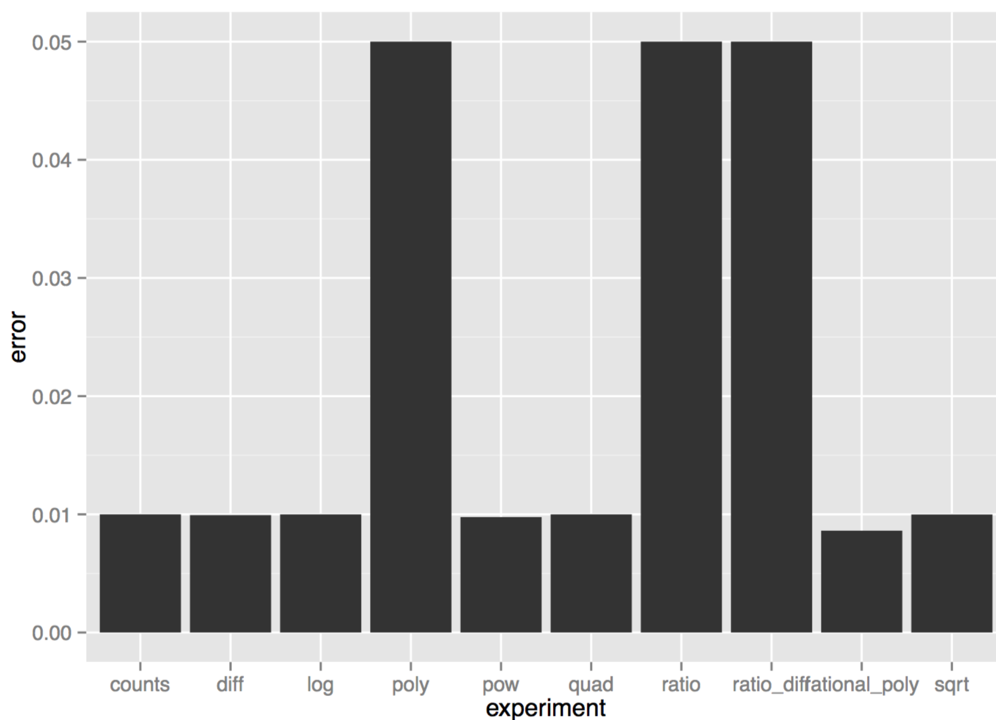

Figure 6. Engineered features for deep learning. Reprinted from An Empirical Analysis of Feature Engineering for Predictive Modeling (p.3), by J. Heaton. Copyright 2016 by IEEE. Reprinted with permission.

The proposed research will make use of the RMSE and multi log-loss error functions. RMSE will be used for all regression problems and multi log-loss will be used for classification. RMSE (McKinney, 2012) is given in *Equation 6* and multi log-loss is given in *Equation 7*.

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

Equation 6. Root mean square error (RMSE)

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

Equation 7. Multi log loss

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

Finding Candidate Real-World Datasets

Ultimately, the utility of this research will be determined by the ability of the developed algorithm to find useful engineered features in real world datasets. Several datasets from the University of California Irvine (UCI) machine learning repository will be used. Datasets that have the following attributes will be favored:

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

The following five datasets appear to be good candidates for this research:

- Adult dataset
- Wine dataset
- Car evaluation dataset
- Wine quality dataset
- Forest fires

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

Milestones

This section details plans to complete the research in a timely manner. The following table shows how the research is broken into a total of 7 tasks to be accomplished over 21 weeks.

#	Description	Duration
1	Setup necessary environments: install deep learning packages, obtain datasets.	1 week
2.	Write software to generate test datasets for feature engineering.	1 week
3.	Create a deep neural network objective function that will evaluate an engineered feature.	1 week
4.	Create initial genetic programming algorithm capable of engineering features.	1 week
5.	Begin experimentation iterations on generated datasets. Three iterations are planned. Each iteration consists of: • Plan features to engineer • Create test dataset • Test genetic programming algorithm • Evaluate results, see what engineered features found	6 weeks total (3 weeks/iteration)
6.	Begin experimentation iterations on real-world datasets	5 weeks total (1 week/dataset)

7.	Write final report	6 weeks total

Table 1. Project schedule

These times are only estimates. Some adjustments will likely be necessary as the project progresses. *Figure 7* shows the project flowchart. The two iteration cycles, for generated and real-world data are shown as the two loops. The following sections describe each of these tasks in greater detail.

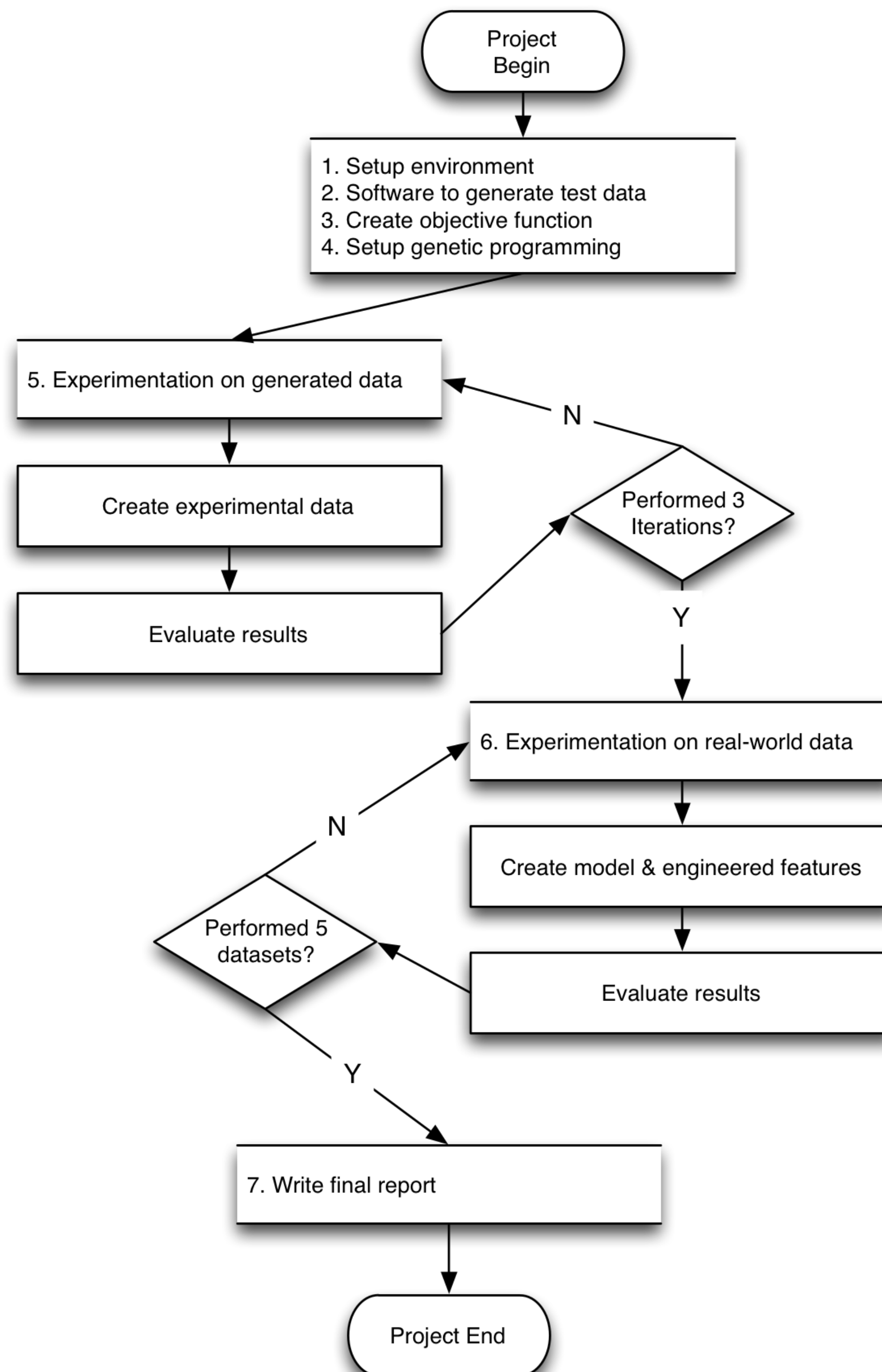

Figure 7. Project flowchart

Project Setup and Creation of Objective Function

The objective function is critical for any project that makes use of an evolutionary algorithm. The objective function will be created in the Java programming language (Arnold, Gosling, & Holmes, 1996). The objective function will be tested using the same datasets as used by Heaton (2016) and should obtain the same RMSE values as the paper. Some analysis will be performed on the minimal complexity and training iterations needed to replicate the results of the paper. The objective function should execute within 10-30 seconds on a single core for it to be effective. A 30 second objective function, with a population size of 1,000, would take approximately (30*1000)/8 = 3,750 (a little over an hour) seconds to execute a single epoch on an 8 core machine. These tasks are estimated at three weeks total, one week to setup environments, one week to create test data, and an additional week to build the objective function.

Setup Genetic Program

This research will make use of the genetic programming algorithms provided by the Encog Machine Learning Framework (Heaton, 2015). Encog contains an advanced genetic programming algorithm that the author developed as prior research to this dissertation. Several palettes will be created that provide the necessary operators to be used for the engineered features. The previously discussed multi-objective function will be created to score engineered features both on their conformance to the desired equation profile, as well as ability to achieve a better loss function result than the control neural network.

Setting up the genetic programming environment might normally take more time than scheduled above. However, the author has extensive experience with the Encog

framework and has helped many in the open source community to adapt problems Encog-based programs to genetic programming. This task is estimated at one week.

Experiment with Generated Data

Once the genetic programming and deep learning framework is in place, experimentation must be performed with generated data. Datasets will be created where the outcome is closely correlated to an engineered feature. The algorithm created in tasks 3&4 will be tested with these datasets to see if the proposed algorithm can discover the underlying engineered feature. These iterations are a sort of sanity check to make sure that the proposed algorithm is capable of finding engineered features that are known to exist and are useful. The operator palette and genetic programming algorithm training parameters will be adjusted at this point to achieve better results.

This round of experiments will be iterative. Each iteration will try a batch of generated data designed to find a variety of features. Data will be generated using simple prototype scripts written in the Python programming language (Van Rossum, 1995). During each iteration the performance of the objective function will be improved. Each iteration is planned to take two weeks, and a total of three iterations are planned. This portion of the research will take six weeks in total.

Experiment with Actual Data

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 dataset. The proposed algorithm developed in the previous iterations will be tested on at least 5 datasets from the UCI Machine Learning Repository (Newman & Merz, 1998). Further

adjustments will likely be made to the proposed algorithm as it is executed against the real-world data. Considerable computation time will likely be needed to sufficiently explore the search space. Additional time will be needed to prepare the datasets for prediction by a neural network. The author has considerable experience preparing datasets for predictive modeling. It is estimated that a week will be needed per dataset. The goal is to evaluate 5 datasets, requiring a total of 5 weeks to complete this task.

Write Final Report

The final dissertation report will be drafted and updated as the tasks progress. However, it will be necessary to spend time editing, finalizing, and tweaking the ultimate dissertation report. This task will include preparing the source code, tables, graphs and other supporting elements of the final dissertation report. The estimated timeframe for this task is approximately 6 weeks.

After completion of this project the author plans to refine the final dissertation report and submit it as an academic paper to a journal or conference. At this point the source code necessary to reproduce this research will be placed on the author's GitHub (http://www.github.com/jeffheaton) repository. For reasons of confidentiality, the source code will not be publicly distributed prior to formal publication.

Resources

The hardware and software components necessary for this research are all standard and readily available common off-the-shelf personal computer system components and software. The author has access to two quadcore Intel I7 Broadwell equipped machines with 16 gigabytes of RAM each. These systems will be used to perform the majority of

computations needed to support this research. If additional processing power is required the author will make use of Amazon AWS virtual machines.

The Java programming language (Arnold et al., 1996) will be used to implement the code necessary to complete this research. The Java 8 version (JDK 1.8) will be used. In addition, Python 3.4 (Van Rossum, 1995) will be used in conjunction with Scipy (Jones, Oliphant, Peterson, & al., 2001), scikit-learn (Pedregosa et al., 2011), Theano (Bastien et al., 2012; Bergstra et al., 2010) and Lasange (Dieleman et al., 2015) 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 be used for 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 research has written much of the code behind Encog and has extensive experience with the Encog framework.

The required equipment is currently available to the author 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 there are issues with the software or implementation of the various components. There is currently no anticipated need to perform interaction with end users, study participants or prior testing data 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, the author will pay them. The author has a 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.

References

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.

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. (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).

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.

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., & 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.

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.

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.

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.

Dieleman, S., Schlüter, J., Raffel, C., Olson, E., Sønderby, S. K., Nouri, D., . . . Heilman, M. (2015). Lasagne: first release.

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

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.

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.

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

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. E., 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. E., & 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*.

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/

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.

Mabu, S., Chen, C., Nannan, L., Shimada, K., & Hirasawa, K. (2011). An intrusion-detection model based on fuzzy class-association-rule mining using genetic network programming. *Transactions on. System Management and Cybernetics, Part C, 41*(1), 130-139.

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).

Mladenić, D. (2006). *Feature selection for dimensionality reduction*. Paper presented at the Proceedings of the 2005 International Conference on Subspace, Latent Structure and Feature Selection, Bohinj, Slovenia.

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

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

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.

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.

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

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.

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).

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.

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

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, 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.

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).