# Performance Effects of Diverse Data Sets on Feed-Forward Neural Networks with Backpropagation, a Genetic Algorithm, Differential Evolution, and Particle Swarm Optimization

**Max Hymer** HYMERMAX1@GMAIL.COM
*Department of Electrical and Computer Engineering*
*Montana State University*
*Bozeman, MT 59715, USA*

**Michael Downs** MICHAELD20312@GMAIL.COM
*Division of Computer Science*
*Montana State University*
*Bozeman, MT 59715, USA*

## Abstract

In this project, we explore multiple Feed-Forward Neural Networks to find how diverse data sets affect the performance of these types of algorithms. We chose Backpropagation, Genetic with Real-Valued Chromosomes, Differential Evolution, and Particle Swarm Optimization algorithms to adjust the neurons within our networks using varied numbers of hidden layers to find how diverse data sets, the number of hidden layers, and the type of neuron update algorithm affect the performance of this type of algorithm. Three of these datasets were classification datasets and three were regression datasets. These datasets were varied in their feature counts, feature ranges, number of classes, target value ranges, and normality. We used Stratified 10-Fold Cross-Validation in order to train, tune, and test our models with accurate representations of these data sets. To evaluate our algorithms, we used 0/1 Loss for classification datasets and MSE for regression datasets. We found that the Genetic, Differential Evolution, and Particle Swarm Optimization algorithms performed better than the Backpropagation algorithm across all data sets. We believe we observed this phenomenon because Backpropagation is a gradient descent algorithm, which is prone to getting stuck in local minima, while the other three algorithms use population-based, nature-inspired weight update methods that aren't prone to gradient-based issues. We found that the Breast Cancer Identification dataset was the highest performing classification dataset across all algorithms, likely due to its large size and balanced data. All of the regression datasets performed relatively well; one data set didn't stand out as the best among them. We did not observe a correlation between average convergence rate and performance; the metric appeared to be tied to the chosen algorithm more than the result for each test. However, it can be noted that Backpropagation generally had a higher average convergence rate and poorer results than the other neuron update algorithms. So, we observed that population-based algorithms require a runtime trade-off for better results. The number of hidden layers appeared to have little impact on performance across all algorithms for classification data sets. However, for regression data sets, performance generally increased as the number of hidden layers increased.

**Keywords:** Neural Network, Backpropagation, Genetic Algorithm, Differential Evolution, Particle Swarm Optimization, Cross-Validation, Regression, and Classification

## 1 Introduction

Recently, there has been a huge jump in substantial findings related to neural networks and deep learning; they are cutting-edge tools that help us predict complicated relationships and find complex patterns within data. Deep learning is used in image recognition, natural language processing, speech recognition, autonomous vehicles, medical diagnosis, financial services, robotics, and more.

With such a real impact of this technology on our lives, it is important to understand and research how it can be used as a tool to better the world we live in. One of the most important decisions for correctly implementing these techniques is to choose the correct model. In this experiment, we aim to help with this decision and contribute to the ongoing research surrounding neural networks by addressing how diverse data sets, the type of neuron update algorithm, and the number of hidden layers within the model affect its performance. Additionally, we want to find how the results from population-based algorithms compare to those from a gradient-based backpropagation algorithm. We hypothesize that the population-based neuron update algorithms will be superior to the gradient-based backpropagation algorithm across all data sets and hidden layer sizes because they explore far more solutions, which prevents them from getting locked into local minima and suboptimal solutions. They are also less sensitive to gradient-based issues like gradient vanishing or exploding. We predict that the Breast Cancer Identification data set will be the best performing classification set across all models because it is large, balanced, and low-noise. Based on our findings in our previous work with Backpropagation, we think that all regression data sets will perform fairly well (without a standout best set) as they seem to correlate well with the neural network framework. We hypothesize that the population-based algorithms will have a lower average convergence rate than the Backpropagation algorithm. So, we believe that it will be tied to performance in that way. However, we are skeptical that we will be able to derive much from observing average convergance rate, especially among classification data sets, as this was the case with our previous findings. Lastly, we believe that two hidden layer models will perform the best for regression data sets while the number of hidden layers will have little affect on classification data sets, again due to our previous finding. All aforementioned findings can be found in more detail in our paper titled Performance Effects of Diverse Data Sets on Feed-Forward Neural Networks with Backpropagation.

## 2 Software Design

Here we will explain our approach to the project software. This means we will go into the significance of our program design, the reasons behind our evaluation metrics, how our learning algorithms work, and a thoughtful explanation of the way it all works together.

### 2.1 Drivers and Helper Classes

We created two unique drivers for each data set that read in the corresponding data set, performed unique preprocessing based on the needs of that data set, and called methods to perform the rest of our experiments. For tuning drivers, a method to extract 10% of the stratified data is called. Then, the remaining data is split into 10 stratified chunks. For testing drivers, the data is just split into 10 stratified chunks. The training and testing data sets for a fold are created and Min-Max Normalized. For classification data set drivers, the training data set outputs (labels) are one-hot encoded. Then, the relevant neuron update algorithm and neural network(s) are initialized and trained with the Min-Max Normalized training data. One final forward pass of the final neural network is performed with the Min-Max Normalized test data. The 0/1 Loss or Mean Squared Error for the fold is calculated from the predictions this forward pass returns. For classification data set drivers, the returned raw percentages are decoded into a predicted class before 0/1 loss is calculated. The test instance, prediction, actual value, 0/1 Loss or MSE, and Average Convergence Rate (if applicable) are printed for the fold. Once the driver iterates through all of the folds, the average MSE or 0/1 Loss and Average Convergence Rate (if applicable) across all of the folds is calculated and printed. The aforementioned Min-Max Normalization, Stratified 10-fold cross-validation (including the stratified 10% extraction), One-Hot Encoding, and MSE calculation

happen within separate helper classes. The Average Convergence Rate calculation happens within the relevant neural network or neuron update algorithm.

### 2.2 Neural Network Algorithm with Backpropagation

We discussed our implementation of this algorithm in great detail in our paper titled Performance Effects of Diverse Data Sets on Feed-Forward Neural Networks with Backpropagation under section 2.2 Neural Network Algorithm. As we are using the same implementation, to keep this paper brief, we will not discuss it in detail within this paper. So, please refer to the above paper for details on our implementation of this algorithm.

### 2.3 Neural Network Algorithm with a Genetic Algorithm with Real-Valued Chromosomes

The neural network first initializes weights using Xavier Initialization (within a weight matrix)

$$W \sim \mathcal{N}\left(0, \left(\sqrt{\frac{1}{n_{in} + n_{out}}}\right)^2\right)$$

where $n_{in}$ is the number of inputs to the layer and $n_{out}$ is the number of outputs for the layer for each layer to ensure that each neuron computes a unique output and contributes differently to the learning process. We chose Xavier initialization because it works well with sigmoid activation functions, like the logarithmic function we chose. Each weight matrix contains the connection weights between each node in the current layer and each node in the next layer. If there are $n$ input features, $h_1$ nodes in the first hidden layer, $h_2$ nodes in the second hidden layer, $h_f$ nodes in the final hidden layer, and $o$ output features, the weight matrix between the input layer and the first hidden layer will have $n \times h_1$ entries, the weight matrix between the first hidden layer and the second hidden layer will have $h_1 \times h_2$ entries, and the weight matrix between the final hidden layer and the output layer will have $h_f \times o$ entries. Each weight in these matrices represents the strengths of the connection between a node in the first layer and a node in the second layer (e.g. each weight in the matrix between the input layer and the first hidden layer represents the strength of the connection between an input feature and a node in the first hidden layer). We initialized biases with Small Random Initialization

$$b \sim \mathcal{N}(0, 0.01^2)$$

so that our model could fit data effectively; these biases shift the activation function up or down which allows the model to learn relationships that don't pass through the origin. Then, the neural network begins the training loop, starting with a forward pass over the data. To start the forward pass, each neuron in a layer computes the weighted sum of its inputs. This weighted sum, denoted as $z$, is calculated as $z = W \cdot x + b$ where $W$ is the weight matrix for the layer, $x$ is the input vector (either from the previous layer or the original input to the network), and $b$ is the bias vector for the neuron in the layer. So, $W \cdot x$, a weighted sum of inputs for a neuron, plus $b$, the bias term, produces the net input to each neuron in the layer. Then, the sigmoid function (a.k.a. the logistic activation function) is applied using

$$a = \sigma(z) = \frac{1}{1 + e^{-z}}$$

where $a$ is the output after applying the activation function and $\sigma(z)$ is the sigmoid function, which squashes the weighted sum $z$ into a value between 0 and 1. This introduces non-linearity into the

model, which allows the neural network to model complex, non-linear relationships in the data. This is the crux of the algorithm; it would simply be a series of linear transformations without it. This process continues through all of the hidden layers until the output layer is reached. For this output layer, a different activation function is chosen based on whether model's data set is a classification set or a regression set. For classification sets, we used the softmax activation function

$$a_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

where $a_j$ is the output for the $j^{th}$ class, $z_j$ is the weighted sum for that class, and $K$ is the number of classes. For regression sets, we used a linear activation. Since our weighted sum $z$ provides a linear output, it becomes our linear activation. So, regression sets essentially just don't use an activation function in the output layer. After this forward pass, our neural network uses a genetic algorithm with real-valued chromosomes to adjust our weights based on loss between the predicted output and the actual target value. But, unlike backpropagation, it doesn't adjust the weights within a single network. Instead, it evaluates the fitness of the given network and uses a selection process to select the most fit networks to survive. First, this algorithm computes the loss for the network's predictions. For regression, it uses negative Mean Squared Error (MSE) Loss

$$-MSE = -\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

where $y_i$ is the actual value, $\hat{y}_i$ is the predicted output, and $n$ is the number of samples. For classification, it uses negative Cross-Entropy Loss

$$\text{-Cross-Entropy Loss} = \sum_{i=1}^{N} y_i \log(\hat{y}_i)$$

where $y_i$ is the actual label and $\hat{y}_i$ is the predicted probability for class $i$. We are using negative MSE and Cross-Entropy Loss because networks with higher fitness scores are considered fitter and therefore more likely to be selected for reproduction in the next generation. So, since we want networks with low MSE or Cross-Entropy Loss, we use the negative of these loss functions. Then, it selects which networks make it to the next generation. We used Roulette Wheel Selection to achieve this. This method first assigns each network a probability of selection proportional to its fitness. So, networks with higher fitness scores are more likely to be selected. Then, figuratively, it spins a roulette wheel where each segment's size is proportional to a network's fitness score and determines the parent networks from a spin of this wheel. We used this method as it ensures diversity in the population as less fit networks still have a chance of being selected. The next step is crossover, which is a process inspired by biological reproduction. It combines the genetic material of two parent chromosomes to produce one or more offspring with the goal of combining weights from both parents to produce a better-performing offspring. We used uniform crossover, which selects each weight randomly from either parent with a fixed probability which we tuned. This method is simple, yet still highly effective, at introducing variation into the population. So, it is used to explore new regions of the solution space that neither of the parents could reach individually. Next, mutation, which is a process that introduces random changes to some of the weights in the offspring networks, occurs. This process is designed to prevent the algorithm from getting stuck in suboptimal local minima by maintaining genetic diversity within the population. It involves adding a small random value (drawn from a Gaussian distribution with a mean of zero and a small standard deviation) to a selected weight. The mutation rate, which is each weight's small

probability of being selected for mutation, determines how many weights will be updated with this method. This rate is a value we tuned. This again expands the solution space beyond the scope visible with simple crossover. Lastly, the process of elitism is applied. This ensures that the best solution from the current generation is preserved in the next generation with no changes, which improves the quality of the population. It works by selecting the best-performing chromosome (neural network with the highest fitness score). Then, it puts this chromosome directly through to the next generation, bypassing crossover and mutation. This way, the best solution is still kept while new solutions are explored through crossover and mutation. After this final step in the training loop is completed, the loop repeats until the fitness scores converge (reach a point where they no longer change significantly between generations). So, we did not define a fixed number of generations; we ran our algorithm as many generations as it took for the fitness scores to converge. We believe that this is the most thorough and accurate way to train our neural network because it allows the algorithm to run until ultimate completion. The neural network returns either a single regression output or an array of class probabilities depending on whether the relevant data set is regression or classification.

### 2.4 Neural Network Algorithm with Differential Evolution

This algorithm works the same way as the GA algorithm until the forward pass is complete. Then, it begins its initialization process. This involves creating an initial population of potential solutions. Each solution, called an individual, consists of a full set of neural network parameters, including the weights and biases that will be used in training. These parameters were initialized through the same method used with the GA algorithm (Xaiver Initialization for weights and Small Random Initialization for biases) as the underlying neural network framework was the same. Once this step was complete, mutation began. This involves creating a new solution, called a "mutant vector," for each individual in the population. To generate this mutant vector $v_i$ for the $i_{th}$ individual $x_i$, three distinct solutions (that must differ from $x_i$) $x_a$, $x_b$, and $x_c$ are randomly selected from the population. Then, the mutant vector $v_i$ is calculated using the formula

$$v_i = x_a + F \times (x_b - x_c)$$

where $F$ is a scaling factor, usually between 0 and 1, which controls the magnitude of the difference between $x_b$ and $x_c$. This scaling factor was a hyperparameter we tuned. It introduces variation in the population which, again, allows the algorithm to thoroughly explore the search space. The difference $(x_b - x_c)$ produces a directional vector that highlights how one individual differs from another, and $F$ scales this difference. Adding this to $x_a$ creates a new, mutated solution $v_i$. Again, as in the GA algorithm, this process is designed to help the algorithm escape suboptimal local minima. Next, the crossover step is used to combine the mutant vector $v_i$ with the current individual $x_i$ to produce a new offspring $u_i$, ensuring that the offspring inherit traits from both the current individual and the mutant vector. For each component $j$ of the offspring $u_i$, a uniformly distributed random number $\text{rand}_j(0, 1)$ is generated. If this number is less than or equal to the crossover probability CR, the component from the mutant vector $v_i$ is chosen. Otherwise, the component from $x_i$ is retained. Additionally, a randomly chosen index $j_{\text{rand}}$ ensures that at least one component from $v_i$ is included in $u_i$, even if $\text{rand}_j(0, 1) > \text{CR}$ for all components. This crossover probability was tuned as well. Then, the selection step, which evaluates the fitness of the offspring, occurs. Just like in the GA algorithm, the negative MSE was used to evaluate fitness for regression data sets while the negative Cross-Entropy Loss was used for classification data sets. If the offspring $u_i$ has a higher fitness score than the the original individual $x_i$, $u_i$ replaces $x_i$ in the next generation. This way, the population will improve in quality over time as the most fit individuals are kept

and the less fit ones are discarded. After this final step in the training loop is completed (again just like in the GA algorithm), the loop repeats until the fitness scores converge (reach a point where they no longer change significantly between generations). Again, we did not define a fixed number of generations; we ran our algorithm as many generations as it took for the fitness scores to converge. And, again, the neural network returns either a single regression output or an array of class probabilities depending on whether the relevant data set is regression or classification.

### 2.5 Neural Network Algorithm with Particle Swarm Optimization

This algorithm works the same way as the GA and DE algorithms (aside from initialization, which uses some algorithm-specific hyperparameters along with the basic neural network parameters) until the forward pass is complete. Then, the Particle Swarm Optimization (PSO) algorithm is used to optimize the neural network's weights. This algorithm, inspired by the social behavior of a flock of birds, employs a population of particles that "fly" through the search space to find the optimal weights for the network. Each particle represents a potential solution, which is set of weights in this case. The "position" of each particle defines the current set of weights while the "velocity" of each particle determines the direction and speed at which a particle searches for a better solution. Just like with the GA and DE algorithms, this algorithm starts with an initialization process. For PSO, this process involves randomly initializing each particle's position (within the search space) and velocity (with the same dimensionality as the position). Then, similarly the other algorithms, the fitness of each particle is assessed with either negative MSE (for regression data sets) or negative Cross-Entropy Loss (for classification data sets). Each particle tracks a personal best, $p_i$, which is the best position (highest fitness) the particle has visited based on its own experience. It also tracks a global best, $g$, which is the best position (highest fitness) found by any particle in the swarm. Since this value is the most successful observed solution, it is used as a guide for the rest of the swarm. Velocity and position updates, which are based on these metrics, happen next. The equation for these updates is

$$v_i = wv_i + c_1 r_1(p_i - x_i) + c_2 r_2(g - x_i)$$

where $v_i$ is the velocity of particle $i$, $w$ is the inertia weight, which controls the influence of the previous velocity, $c_1$ and $c_2$ are the cognitive and social acceleration coefficients, respectively, $r_1$ and $r_2$ are random values between 0 and 1, introducing stochasticity to prevent premature convergence and ensure diverse exploration, $p_i$ is the particle's personal best position, $g$ is the global best position, and $x_i$ is the current position of the particle. Within this equation, a higher $w$ encourages exploration, while a lower $w$ promotes exploitation (focusing on promising areas) and $c_1$ and $c_2$ determine how strongly a particle is pulled towards its personal best and the global best. The inertia weight and the cognitive and social acceleration coefficients were tuned here. To prevent particles from moving too quickly, velocity clamping was applied with

$$v_i \leftarrow \begin{cases} v_i & |v_i| < v_{\max} \\ v_{\max} & |v_i| \geq v_{\max} \end{cases}$$

where $v_{\max}$ is the predefined maximum velocity. This ensures that each component of $v_i$ stays below $v_{max}$. This way, the velocities were not able to grow to such a high level that they no longer had any real meaning. The value of $v_{max}$ was tuned here. The position of each particle is then updated using

$$x_i = x_i + v_i$$

in order to shift the particle to a (hopefully) more optimal position in the solution space. Unlike with the previous two algorithms, we did not run our particle swarm optimization algorithm until

convergence. Instead, we ran it for a fixed number of generations. We felt that attempting to train this model to convergence required too much unnecessary complexity without enough upside, so we felt that using a fixed number of generations was the best solution. So, after this final step in the training loop is completed, the loop repeats until the fixed number of generations is reached. The neural network returns either a single regression output or an array of class probabilities depending on whether the relevant data set is regression or classification, just like with the other algorithms.

## 3 Experimental Approach

Our test strategy was a mostly uniform process across the six datasets. Each dataset was modeled with each neuron update algorithm with zero, one, and two hidden layers with the best hidden layer sizes from our Backpropagation testing. First, we performed data imputation. This was only necessary on the Breast Cancer Identification data set. To do this, we replaced instances of "?" in the data with a random integer with a value of 1-10 as that is the range of the data and we wanted to keep our data imputation approach simple. Then, we performed 10-Fold Stratified Cross-Validation. For tuning, this strategy involves taking out a stratified 10% of the data and using it as the test fold, splitting the remaining data into 10 stratified folds, removing 1 fold, and combining the remaining 9 folds into training data until each fold has had a turn being the removed fold. For testing, this strategy involves splitting the data into 10 stratified folds, removing 1 fold to use as the test fold, and combining the remaining 9 folds into training data until each fold has had a turn being the test fold. We Min-Max Normalized both the training and testing folds to make our data compatible with our neural networks. Next, we initialized the appropriate neuron update algorithm with the appropriate hyperparameters. We tuned as much of the typical range for each hyperparameter as time allowed, however, the long runtime of the population-based algorithms was a major tuning limitation. It is also important to note that we only tuned our population-based algorithm's hyperparameter values for two hidden layer networks as we assumed these networks to have the best performance based on our results from Backpropation testing. One and zero hidden layer networks are assumed to have the same tuning values as a two layer network, which is definitely a limitation we will discuss later in the paper. We then trained and tested each neuron update algorithm with the best tuned hyperparameters we found by averaging the algorithm's performance of across ten test folds for a two hidden layer, one hidden layer, and no hidden layer network. This is the average that is included under the 0/1 or MSE column in our results.

## 4 Results

This is our results section where we will represent the data that was found from our experiment. Reflection on the data and the limitations to our testing approach will be expressed in the sub-sections below the final data figures. Abbreviations are used within our results tables: 0/1 is the 0/1 Loss, MSE is the Mean Squared Error, ACR is the Average Convergence Rate, Backprop is Backpropagation, GA is Genetic Algorithm with Real-Valued Chromosomes, DE is Differential Evolution, and PSO is Particle Swarm Optimization.

### 4.1 Breast Cancer Identification (Classification)

**Two Hidden Layers: [6, 4]**

| Model | 0/1 | ACR |
|---|---|---|
| Backprop | 0.3461 | 0.0044 |
| GA | 0.0987 | 0.0003 |
| DE | 0.0373 | 0.0009 |
| PSO | 0.0416 | NA |

**One Hidden Layer: [6]**

| Model | 0/1 | ACR |
|---|---|---|
| Backprop | 0.3378 | 0.0039 |
| GA | 0.0387 | 0.0003 |
| DE | 0.0387 | 0.0001 |
| PSO | 0.0444 | NA |

**No Hidden Layers**

| Model | 0/1 | ACR |
|---|---|---|
| Backprop | 0.2833 | 0.0123 |
| GA | 0.0359 | 0.0004 |
| DE | 0.1317 | 0.0023 |
| PSO | 0.1254 | NA |

Figure 1: Breast Cancer Identification Results Tables with the best tuned hyperparameters: a learning rate of 0.0001 and momentum off for the Backprop model, a population size of 100, a mutation rate of 0.01, and a crossover rate of 0.7 for the GA model, a population size of 50, a binomial crossover probability of 0.7, and a scaling factor of 0.3 for the DE model, and 100 particles, a maximum iterations value of 200, an inertia of 1.0, a cognitive update rate of 2.5, a social update rate of 2.0, and a maximum velocity of 0.1 for the PSO model

### 4.2 Soybean Identification (Classification)

**Two Hidden Layers: [33, 15]**

| Model | 0/1 | ACR |
|---|---|---|
| Backprop | 0.6909 | 0.0106 |
| GA | 0.2400 | 0.0015 |
| DE | 0.2400 | 0.0003 |
| PSO | 0.2400 | NA |

**One Hidden Layer: [33]**

| Model | 0/1 | ACR |
|---|---|---|
| Backprop | 0.7159 | 0.0024 |
| GA | 0.2650 | 0.0022 |
| DE | 0.2650 | 0.0000 |
| PSO | 0.2650 | NA |

**No Hidden Layers**

| Model | 0/1 | ACR |
|---|---|---|
| Backprop | 0.7545 | 0.0049 |
| GA | 0.2400 | 0.0024 |
| DE | 0.2650 | 0.0000 |
| PSO | 0.2400 | NA |

Figure 2: Soybean Identification Results Tables with the best tuned hyperparameters: a learning rate of 0.001 and a momentum coefficient of 0.9 for the Backprop model, a population size of 50, a mutation rate of 0.1, and a crossover rate of 0.8 for the GA model, a population size of 100, a binomial crossover probability of 0.9, and a scaling factor of 0.3 for the DE model, and 100 particles, a maximum iterations value of 100, an inertia of 0.7, a cognitive update rate of 2.5, a social update rate of 2.5, and a maximum velocity of 0.1 for the PSO model

### 4.3 Glass Identification (Classification)

**Two Hidden Layers: [6, 4]**

| Model | 0/1 | ACR |
|---|---|---|
| Backprop | 0.6308 | 0.0051 |
| GA | 0.6001 | 0.0003 |
| DE | 0.5689 | 0.0001 |
| PSO | 0.6583 | NA |

**One Hidden Layer: [6]**

| Model | 0/1 | ACR |
|---|---|---|
| Backprop | 0.6269 | 0.0055 |
| GA | 0.4208 | 0.0002 |
| DE | 0.6633 | 0.0001 |
| PSO | 0.6621 | NA |

**No Hidden Layers**

| Model | 0/1 | ACR |
|---|---|---|
| Backprop | 0.6308 | 0.0077 |
| GA | 0.4417 | 0.0003 |
| DE | 0.6072 | 0.0001 |
| PSO | 0.5846 | NA |

Figure 3: Glass Identification Tables with the best tuned hyperparameters: a learning rate of 0.001 and momentum off for the Backprop model, a population size of 50, a mutation rate of 0.01, and a crossover rate of 0.7 for the GA model, a population size of 100, a binomial crossover probability of 0.9, and a scaling factor of 0.3 for the DE model, and 100 particles, a maximum iterations value of 200, an inertia of 1.0, a cognitive update rate of 2.5, a social update rate of 2.0, and a maximum velocity of 0.1 for the PSO model

### 4.4 Computer Hardware (Regression)

**Two Hidden Layers: [4, 2]**

| Model | MSE | ACR |
|---|---|---|
| Backprop | 0.0567 | 0.0490 |
| GA | 0.0161 | 0.0001 |
| DE | 0.0135 | 0.0006 |
| PSO | 0.0181 | NA |

**One Hidden Layer: [4]**

| Model | MSE | ACR |
|---|---|---|
| Backprop | 0.0523 | 0.0917 |
| GA | 0.0163 | 0.0001 |
| DE | 0.0172 | 0.0019 |
| PSO | 0.0146 | NA |

**No Hidden Layers**

| Model | MSE | ACR |
|---|---|---|
| Backprop | 0.0949 | 0.0085 |
| GA | 0.0161 | 0.0001 |
| DE | 0.0148 | 0.0003 |
| PSO | 0.0149 | NA |

Figure 4: Computer Hardware Results Tables with the best tuned hyperparameters: a learning rate of 0.01 and momentum off for the Backprop model, a population size of 50, a mutation rate of 0.1, and a crossover rate of 0.8 for the GA model, a population size of 50, a binomial crossover probability of 0.7, and a scaling factor of 0.5 for the DE model, and 100 particles, a maximum iterations value of 50, an inertia of 0.7, a cognitive update rate of 1.5, a social update rate of 1.5, and a maximum velocity of 0.1 for the PSO model

### 4.5 Abalone (Regression)

**Two Hidden Layers: [5, 3]**

| Model | MSE | ACR |
|---|---|---|
| Backprop | 0.0251 | 0.2243 |
| GA | 0.0192 | 0.0001 |
| DE | 0.0281 | 0.0022 |
| PSO | 0.0170 | NA |

**One Hidden Layer: [5]**

| Model | MSE | ACR |
|---|---|---|
| Backprop | 0.0421 | 0.0625 |
| GA | 0.0166 | 0.0001 |
| DE | 0.0331 | 0.0036 |
| PSO | 0.0219 | NA |

**No Hidden Layers**

| Model | MSE | ACR |
|---|---|---|
| Backprop | 0.1001 | 0.0139 |
| GA | 0.0420 | 0.0001 |
| DE | 0.0631 | 0.0001 |
| PSO | 0.0495 | NA |

Figure 5: Abalone Results Tables with the best tuned hyperparameters: a learning rate of 0.001 and momentum off for the Backprop model, a population size of 100, a mutation rate of 0.1, and a crossover rate of 0.9 for the GA model, a population size of 50, a binomial crossover probability of 0.9, and a scaling factor of 0.7 for the DE model, and 50 particles, a maximum iterations value of 100, an inertia of 1.0, a cognitive update rate of 0.5, a social update rate of 0.1, and a maximum velocity of 0.1 for the PSO model

### 4.6 Forest Fires (Regression)

**Two Hidden Layers: [6, 4]**

| Model | MSE | ACR |
|---|---|---|
| Backprop | 0.0239 | 0.3167 |
| GA | 0.0249 | 0.0000 |
| DE | 0.0240 | 0.0005 |
| PSO | 0.0239 | NA |

**One Hidden Layer: [6]**

| Model | MSE | ACR |
|---|---|---|
| Backprop | 0.0251 | 0.3439 |
| GA | 0.0251 | 0.0000 |
| DE | 0.0246 | 0.0014 |
| PSO | 0.0243 | NA |

**No Hidden Layers**

| Model | MSE | ACR |
|---|---|---|
| Backprop | 0.0986 | 0.1480 |
| GA | 0.0245 | 0.0002 |
| DE | 0.0245 | 0.0001 |
| PSO | 0.0250 | NA |

Figure 6: Forest Fires Results Tables with the best tuned hyperparameters: a learning rate of 0.001 and momentum off for the Backprop model, a population size of 100, a mutation rate of 0.05, and a crossover rate of 0.8 for the GA model, a population size of 100, a binomial crossover probability of 0.7, and a scaling factor of 0.5 for the DE model, and 100 particles, a maximum iterations value of 100, an inertia of 0.7, a cognitive update rate of 2.5, a social update rate of 2.0, and a maximum velocity of 0.1 for the PSO model

### 4.7 Discussion

Reflecting on our hypothesis concerning population-based algorithm performance in comparison to Backpropagation algorithm performance, we were correct; the population-based algorithms performed better across all data sets and model architectures. Overall, for our regression data sets, our population-based algorithms performed much better with no hidden layers compared to our Backprop no hidden layer performance, which was interesting. This is likely because Backprop updates weights by propagating error backwards through network layers, so it relies heavily on hidden layers while the population-based algorithms do not. It was surprising to see that there was not a clear best population-based algorithm; each algorithm's performance in comparison to the others was very data set and model architecture specific. Overall, the backpropagation algorithms had a much faster convergence rate, which is important when considering run time, but also seems to suggest that a higher average convergence rate is linked to a poorer performance. The Breast Cancer Identification data set was, as we expected, the best performing classification set by far. The regression sets all performed similarly, following in the footsteps of the Backprop algorithm, which is what we expected as well. The classification sets did not have a clear pattern between hidden layer size and performance and the regression data sets generally performed better on two hidden layer models (again as expected).

### 4.8 Limitations

While our experiment produced many valuable results, it also had some limitations. It is important to recognize that our hyperparameter tuning ranges were nowhere near all-encompassing and somewhat arbitrarily chosen. The extreme run time for population-based algorithms exasperated this issue; so the tuning value ranges were especially low for these algorithms. The fact that we only tuned values for two hidden layer networks (for our population-based algorithms) is also a huge limitation, the best hyperparameters for this architecture were not likely the best hyperparameters for the one and no hidden layer architectures. This is important because it means that our hyperparameters were not necessarily the most ideal values to give us the most accurate models. If we had more time to tune more values, we may have been able to make our models perform better and therefore acquire better results. We also only looked at six datasets, and, although diverse, they do not cover anywhere near all possible types of data sets. So, by processing such a small scope of the information that can be processed, our findings likely do not apply to all possible data sets.

## 5 Summary

Through this experiment, we gleaned valuable insight into the appropriate uses of Feed-Forward Neural Networks with Backpropagation, a Genetic Algorithm with Real-Valued Chromosomes, Differential Evolution, and Particle Swarm Optimization. The process of choosing and tuning a learning model for a specific problem is a vital part of approaching a machine learning challenge and achieving success. So, we are glad to have learned about and contributed research to this topic. Within our process, we found that population-based neuron update algorithms are far better than gradient-based neuron update algorithms across all architectures and data sets. But, they use much more processing power and have a much smaller average convergence rate, leading to a far longer runtime. We did not find a best population-based algorithm across all data sets and architectures, so it is clearly important to experiment with different algorithms and architectures under the population-based algorithm umbrella to find the best results for a specific data set.

## 6 Appendix

Max Hymer:
- Paper (Abstract, Introduction, Experimental Approach, Results Tables)
- Code (GA)
- 50% effort

Michael Downs:
- Paper (Software Design, Limitations)
- Code (DE, PSO)
- Video commentary and recording
- 50% effort

Combined Work:
- Paper (Discussion, Summary)