# A Custom Multi-Layer Perceptron Approach: Comparing Adam, Mini-Batch & Stochastic Gradient Descent Optimizers in Diabetes Prediction

*M. Fatih A.*
*xxxxxxx*
*Vrije Universiteit Amsterdam*
*xxxxxx*

*Kayra Ö.*
*xxxxxxx*
*Vrije Universiteit Amsterdam*
*xxxxxx*

*Yusuf A.*
*xxxxxxx*
*Vrije Universiteit Amsterdam*
*xxxxxx*

*Tim v.d. T.V.*
*xxxxxxx*
*Vrije Universiteit Amsterdam*
*xxxxxx*

*Vaamika R.*
*xxxxxxx*
*Vrije Universiteit Amsterdam*
*xxxxxx*

**Abstract—*This project focuses on the development of a neural network in Python from scratch, using only the NumPy library; and on the interpretation of results acquired from the comparison amongst variations of the gradient descent algorithm within the model. More specifically, there are 4 versions of the model, with the two base changes: an (activated/deactivated) AdaM optimizer, and a toggle between stochastic gradient descent (SGD) or mini-batch gradient descent. The model is tasked with binary classification of predicting whether an individual has diabetes or not. When combined with the expertise of a human doctor or diagnostician, this type of model can serve as a powerful decision-support tool, enhancing diagnostic accuracy, identifying at-risk patients earlier, and optimizing treatment plans.***

***Keywords–artificial intelligence, neural network, MLP, diabetes, machine learning, health, AdaM optimizer, optimization methods, binary classification, implementation from scratch, gradient descent methods, numpy***

## I. Introduction

**Artificial Intelligence** has been an impactful innovation in healthcare with its intricate algorithms and various applications **[1]**; such as health information systems, geocoding health data, **predictive modelling and decision support,** and medical imaging etc. **[2]**. One of the promising domains of AI applications is patient data and diagnostics, where it is aimed to help medical researchers effectively analyze and interpret big **datasets** from patients to use them on special AI techniques **[2]**. The capability of processing big datasets make **machine learning models** suitable for this job **[3]**. This project focuses on the development of a **neural network** (more specifically an **MLP**) in Python **from scratch**, *using only the NumPy library*; and on the interpretation of results acquired from the comparison amongst variations of the **gradient descent** algorithm within the model. This results in the research question of '***How do SGD, Mini-Batch, and Adam optimization techniques affect the performance metrics of a custom-implemented MLP model on a diabetes dataset?***'

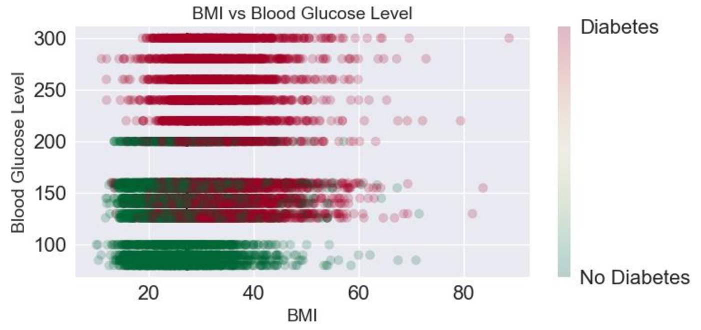
There are 4 versions of the model being implemented, with the two base changes: an (activated/deactivated) **AdaM optimizer**, and a toggle between **stochastic gradient descent (SGD)** or **mini-batch gradient descent**. The model is tasked with **binary classification** of predicting whether an individual has diabetes or not. When combined with the expertise of a human doctor or diagnostician, this type of model can serve as a powerful decision-support tool, enhancing diagnostic accuracy, identifying at-risk patients earlier, and optimizing treatment plans. The prediction is made depending on a set of various features, including individual demographics such as *gender* and *age*, and more specific health conditions such as looking at *BMI*, the presence of *heart disease*, presence of *hypertension*, *blood glucose levels* as well as *Haemoglobin A1c levels*. Machine learning models are becoming increasingly used in medical applications to deliver accuracy and reliability, so it is important to ensure the model's high performance and to know which design choices affect the process in order to build more robust models.

Throughout the development of this project, some implementations/changes were made that were later discarded. Some notable changes are: the dataset used for the model was previously a mushroom dataset, also with a binary classification task of predicting whether a given mushroom is poisonous or edible. This dataset proved too small, and hence was changed. Furthermore, all features within this past dataset were categorical features - for which **one-hot encoding** was implemented; however in the most recent version of the code this is deleted, as we do *not* use one-hot encoding for the newer dataset/features.

## II. Dataset and preprocessing

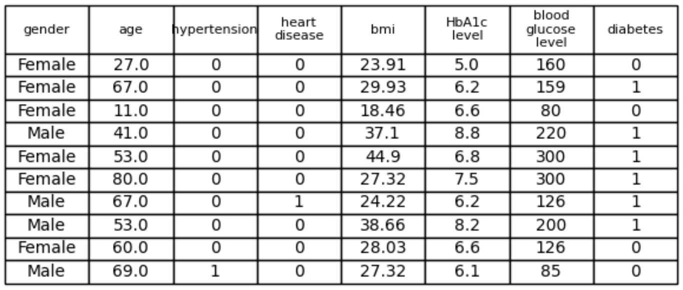
As previously mentioned, the model was trained on a structured dataset focused on diabetes classification. The diabetes dataset was obtained from *Kaggle* **[10]**. Each instance in the dataset represents an individual patient, labeled with either 0 (not diagnosed with diabetes) or 1 (diagnosed with diabetes). Seven input features are used for prediction: *gender*, *age*, *body mass index (BMI)*, presence of *hypertension*, presence of *heart disease*, *blood glucose level*, and *HbA1c percentage*. Most features are numerical, except for *gender*, which is binary categorical (0 or 1) and left unmodified during preprocessing (Since neural networks can handle binary categorical features without the need of one-hot encoding **[4]**).

*Fig. 1: BMI vs Blood Glucose Level in a distribution with and without diabetes.*



For the data preprocessing, the dataset contained high **class imbalance** (there are 92k instances of healthy and 8k instances of ill people) - to combat this, an implementation of **random undersampling** was applied, where the number of healthy instances was lowered to be 8k, to achieve an even split amongst the target class. The dataset was further cleaned up by the **column-wise deletion** of the feature “smoking_history” as it contained missing values, and by the **list-wise deletion** of only two instances as they contained a third option for the categorical feature *gender* (namely “Other”).

*Fig. 2: A snippet of the dataset, with headers provided at the top*



| gender | age | hypertension | heart disease | bmi | HbA1c level | blood glucose level | diabetes |
|---|---|---|---|---|---|---|---|
| Female | 27.0 | 0 | 0 | 23.91 | 5.0 | 160 | 0 |
| Female | 67.0 | 0 | 0 | 29.93 | 6.2 | 159 | 1 |
| Female | 11.0 | 0 | 0 | 18.46 | 6.6 | 80 | 0 |
| Male | 41.0 | 0 | 0 | 37.1 | 8.8 | 220 | 1 |
| Female | 53.0 | 0 | 0 | 44.9 | 6.8 | 300 | 1 |
| Female | 80.0 | 0 | 0 | 27.32 | 7.5 | 300 | 1 |
| Male | 67.0 | 0 | 1 | 24.22 | 6.2 | 126 | 1 |
| Male | 53.0 | 0 | 0 | 38.66 | 8.2 | 200 | 1 |
| Female | 60.0 | 0 | 0 | 28.03 | 6.6 | 126 | 0 |
| Male | 69.0 | 1 | 0 | 27.32 | 6.1 | 85 | 0 |

All numerical features are **normalized** (via z-score normalization, i.e. standardization) to improve training stability, while the *gender* feature remains in its binary form (0 or 1). The data is then split into training, validation, and test sets using an **80/10/10** ratio, respectively. The validation set is used during hyperparameter selection via **grid-search** to identify the best-performing model setup, and only then is the test set used to do a final evaluation of the best model.

*z-score normalization*

$$x_{normalized} = \frac{x - \mu}{\sigma}$$

## III. Methodology

Starting with unpacking the foundation of the custom built model, we will specify all the design features of the model including the choice of each structure. Firstly, the neural network built is a Multi-layer Perceptron (MLP), or also called a **feed-forward neural network**. This is built using **NumPy**, a *Python* library used for numerical computing, that deals with multi-dimensional arrays, allowing us to **vectorize** abstract tasks. The design consists of two main classes, with multiple methods, and five additional functions.

### A. Initialization

Weight initializations play an important role to help the gradients to remain stable during training, particularly when activation functions like **ReLU** and **Sigmoid** have been applied, where poor initialization can lead to common issues such as vanishing gradients and inactive *(dead)* neurons. The implementation uses a custom initialization where the model uses a hybrid implementation between **Xavier** and **He** initialization. Xavier and He initialization are defined as follows:

$$\square\square\square\square\square\square\square = \sqrt{\frac{1}{\square\square\square\square\square+\square\square\square\square\square\square\square}}\ , \quad \square\square = \sqrt{\frac{2}{\square\square\square\_\square\square}}$$

The Xavier initialization works well for the Sigmoid activation function, while the He initialization better suits the ReLU function because of how it zeroes negative values. [6] The following custom-made hybrid function provides the best of both worlds:

$$\square^{(\square)} = \sim\square(0, \sqrt{\frac{2}{\square\square\square\_\square\square + \square\square\square\_\square\square\square}})$$

Where the weights are made sure not to be too small like in Xavier, or too large like in He, keeping the activations in a balanced range.

## B. Baseline

Before training the neural network, we established a **majority class baseline** model that always predicts the most prevalent (target) class in the training data to provide a lower bound reference point for the evaluation.

## C. Layers

The architecture of the neural network is composed of an input layer with size 7 (receives 7 features), a *custom amount of hidden layers*, and the output layer with size 1 (binary classification, so only 1 output).

## D. Grid-Search

The hidden layer design is decided via the validation set and **grid-search**. There are some predefined values of *learning_rate* and *hidden_size* for which the grid-search tries combinations of (namely 0.01, 0.05, 0.001 for *learning_rate*, and [16], [32], [64], [32 16], [64 32], [64 32 16] for the *hidden_size*; where a number represents the neuron amount, and the length of the list represents the amount of hidden layers). Multiple models are created with the combinations of these hyperparameters, and the validation set is used to evaluate the performance of the trained models. The model that gives the best 'metric' (can be changed when being passed down as a parameter, defaults to *accuracy*) is then selected to be the main model.

## E. Training

Each layer receives the inputs, applies weights and biases to the inputs, and transfers the information to the activation function ReLU. ReLU returns the input as it is if it is positive, or a zero if otherwise. This was chosen because it introduces non-linear relationships while still remaining efficient, and avoids issues like vanishing gradients that might occur with other activation functions. The output is then applied to a sigmoid activation function to yield a probability between 0 and 1 for whether the patient has diabetes or not. The ReLU and sigmoid methods are both implemented inside the MLP class, alongside two other methods *ReLU_derivative* and *sigmoid_derivative*, which are methods to compute the derivatives of these activation functions. Moreover, the sigmoid activation function contains a clipping interval in its definition, ensuring no exploding or vanishing gradients.

The training method is designed with a **patience strategy**, where if the loss function has not (significantly) decreased in a number of epochs, the training is terminated to prevent further over-fitting, known as an early stopping **[8].**

In the forward pass, the prediction is computed via the following operations, where □ stands for *activation layers*, □ for *pre-activation layers*, □ for *weight matrices*, and □ for the *bias vector*:

$$\square^{(\square)} = \square^{(\square-1)}\square^{(\square)} + \square^{(\square)}$$
$$\square^{(\square)} = \square\square\square\square(\square^{(\square)}) = \square\square\square(0, \square^{(\square)})$$
$$\square^{(\square)} = \square^{(\square-1)}\square^{(\square)} + \square^{(\square)}$$
$$\square^{(\square)} = \square(\square^{(\square)}) = \frac{1}{1 + \square^{(-\square\square\square\square(\square^{(\square)}, -50, 50))}}$$
$$\underline{\hat{y} = \square^{(\square)}\square\ (0,1)}$$

## F. Loss Function

The loss function is selected to be **binary cross-entropy loss** (BCE), as the task at hand is a binary classification task. The initial implementation contained a mean-squared error (MSE) loss instead of the BCE loss, however BCE is specifically formulated for classification problems in which the model outputs represent probabilities, such as those produced by a sigmoid activation function in the output layer. In contrast to MSE, which measures the average squared difference between predicted probabilities and true labels, BCE provides more appropriate gradient signals for classification. MSE tends to yield smaller gradients when predictions are close to the correct class, which can slow down the learning process; while BCE penalizes confident but incorrect predictions more heavily and generates larger, more informative gradients, particularly during the early stages of training. Below is the formula for BCE, where *l* stands for the loss, and *m* stands for the batch size.

$$\square = -\frac{1}{\square}\sum_{\square=1}^{\square} [\square_{\square}\ \square\square\square(\hat{y}_{\square}) + (1 - \square_{\square})\ \square\square\square(1 - \hat{y}_{\square})]$$

## G. Backpropagation

Once $\hat{y}$ is computed from the forward pass, backpropagation occurs to compute the gradients of the loss. Backpropagation works backwards through each layer, with the use of the chain rule layer by layer, analyzing which weights need to be adjusted for a lower loss yield. There is a variation in the computing, depending on whether the optimizer is toggled on or off. Furthermore, the derivative of the sigmoid allows the first equation to be simplified into a simple residual:

$$\delta^{(\square)} = \frac{\square\square}{\square\square^{(\square)}} = \hat{y} - \square$$

$$\delta^{(\square)} = \square^{(\square+1)}(\square^{(\square+1)})^{\square} \odot \square\square\square\square'(\square^{(\square)})$$

$$\frac{\square\square}{\square\square^{(\square)}} = \frac{1}{\square}(\square^{(\square-1)})^{\square}\square^{(\square)}$$

$$\frac{\square\square}{\square\square^{(\square)}} = \frac{1}{\square}\sum_{\square=1}^{\square} \square_{\square}^{(\square)}$$

*for no-optimizer:*

$$\square^{(\square)} = \square^{(\square)} - \square\frac{\square\square}{\square\square^{(\square)}}$$

$$\square^{(\square)} = \square^{(\square)} - \square\frac{\square\square}{\square\square^{(\square)}}$$

*for optimizer:*

$$\square^{(\square)} = \square^{(\square)} - \square_{\square}\frac{\hat{\square}_{\square}^{(\square)}}{\sqrt{\hat{\square}_{\square}^{(\square)}} + \square} \quad \square^{(\square)}$$

$$= \square^{(\square)} - \square_{\square}\frac{\hat{\square}_{\square}^{(\square)}}{\sqrt{\hat{\square}_{\square}^{(\square)}} + \square}$$

where $\delta$ is the gradient w.r.t. $\square^{(\square)}$, $\square$ is the loss, $\hat{y}$ is the predicted output (probability estimate), $\square$ is the true label (0 or 1), $\square$ is the number of training samples in the batch (i.e. batch size), $\eta$ is the learning rate, $\hat{\square}_{\square}^{(\square)}$ and $\hat{\square}_{\square}^{(\square)}$ are the first and second moment estimates (Adam's moving averages of gradients and squared gradients for weights), $\epsilon$ is a small constant for numerical stability in Adam, and $\odot$ is **element-wise** *(Hadamard)* **product**.

## H. AdaM

The implementation of AdaM (adaptive moment estimation) optimizer follows directly from the original Kingma & Ba paper **[5]**. The specifics of the optimizer, such as *beta1*, *beta2* or *epsilon* are left unchanged. The AdaM optimizer specifically monitors the second raw moment, the average of the gradients squared, and the momentum, which is the exponentially decaying average of the previous gradients, also known as **smoothing** the updates. When there is no optimizer present, a classical gradient descent approach (with versions mini-batch or stochastic) is used. The operations are done in the following way, where $\beta_{\square}$ stands for beta *n*:

*first moment estimate*

$$\square_{\square}^{(\square)} = \square_{1}\square_{\square-1}^{(\square)} + (1 - \square_{1})\square_{\square^{(\square)}}\square$$

*second moment estimate*

$$\square_{\square}^{(\square)} = \square_{2}\square_{\square-1}^{(\square)} + (1 - \square_{2})(\square_{\square^{(\square)}}\square \odot \square_{\square^{(\square)}}\square)$$

*bias correction*

$$\hat{\square}_{\square}^{(\square)} = \frac{\hat{\square}_{\square}^{(\square)}}{1 - \square_{1}^{\square}}, \quad \hat{\square}_{\square}^{(\square)} = \frac{\square_{\square}^{(\square)}}{1 - \square_{2}^{\square}}$$

*parameter update*

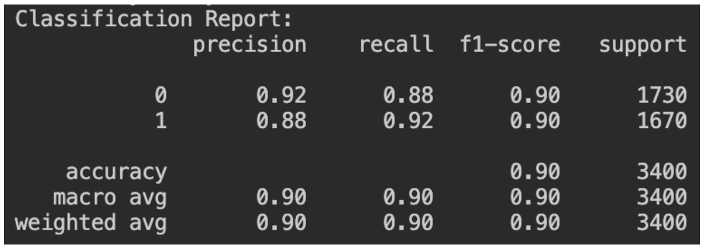
$$\eta_{\square} = \square\frac{\sqrt{1 - \square_{2}^{\square}}}{1 - \square_{1}^{\square}}$$

*(computing of new $\square$ and $\square$ in the backprop. section)*

## I. Evaluation

These optimization strategies were compared under controlled conditions, where tests were conducted with stochastic gradient descent (SGD) and with mini-batch gradient descent, with the AdaM optimizer and without, facilitating 4 tests in total (SGD+AdaM, SGD alone, mini-batch +AdaM, mini-batch alone). The test was conducted 4 times across four random seeds (the same seeds were used for different models, but different seeds were used for different testings) to evaluate how the optimizer and gradient descent implementations affect the classification performance and training. The evaluation of the models were done with the following metrics: *accuracy, F-1 score, precision*, *recall* from the **confusion matrix [7]**. Furthermore, an implementation using the **sk-learn** implementation was done purely to compare metrics with the MLP implementation, yielding the following results that could be used as a reference point for the following *statistical testing* and *evaluation* sections (where 0 stands for a *no diabetes* prediction and 1 stands for a *diabetes* prediction):

*Fig. 3: Classification Report of sk-learns model*



```
Classification Report:
              precision    recall  f1-score   support

           0       0.92      0.88      0.90      1730
           1       0.88      0.92      0.90      1670

    accuracy                           0.90      3400
   macro avg       0.90      0.90      0.90      3400
weighted avg       0.90      0.90      0.90      3400
```

### J. Feature Importance

Finally, a method *get_feature_importance( )* is implemented that quantifies the effect of every input feature by measuring the drop in performance when individual features are shuffled (perturbed) **[9]**. This setup provides insights within the dataset and the field, and could contain implications that could allow benefits in the reproduction of the experiment, or other experiments making use of this dataset.

$$\square\square\square\square\square\square\square\square\square\square\square_{\square} = \square\square\square\square\square\square\square\square\square_{\square\square\square\square\square\square\square\square\square} - \square\square\square\square\square\square\square\square\square_{\square h\square\square\square\square\square\square}$$

### K. Execution

The algorithm is organized around one core function: *run_classification( ),* which handles the entire experiment, with the exception of the baseline model creation. The function starts with data preprocessing, hyperparameter tuning via grid search on the validation set, and the final evaluation on the test set. **[11]** This function takes four parameters, namely *optimizer* (AdaM or none), *gdescent* (mini-batch or SGD), *random_seed* which defaults to none without a parameter, and *metric_for_selection* which selects the metric to compare during grid-search to find the best model. The full representation of the architecture can be seen below:

```
--- CLASS AdamOptimizer
├── __init__
├── initialize
└── update

--- CLASS MLP
├── __init__
├── sigmoid
├── sigmoid_derivative
├── relu
├── relu_derivative
├── binary_cross_entropy
├── forward
├── backward
├── train
├── predict
├── evaluate
└── get_feature_importance

--- FUNCTION preprocess_data
--- FUNCTION run_classification
--- FUNCTION calculate_metrics
--- FUNCTION create_baseline_model

=== MAIN SCRIPT
```

The complete implementation of this research, including the source code and experimental scripts, is available at https://github.com/AresPaleson/MLP-from-scratch.git.

## IV. Statistical Testing

### A. Hypothesis Testing

In order to efficiently assess which out of the 4 optimization strategies has a significant influence on the performance of the 4 models, a number of statistical tests will be performed. Based on the research question, the first statistical analysis performed will be calculating the **mean** and **standard deviation** of each performance metric for each optimization setup. The mean describes the average performance of each model, whereas the standard deviation describes how much the scores observed vary from the mean. These two calculations are significant for this analysis as it measures reliability, where the mean displays the overall performance of the metric, whilst the standard deviation builds on this, and displays how consistent the optimizer is.

Along with this, as values can take small values, it is important to assess whether the optimization strategies have a **significant** influence on the performance. This will be tested efficiently through a more complex statistical test, called the ANOVA test. ANOVA testing is used to analyze significant differences between groups of 3 or more, that utilizes the mean of the data, checking the variation between all groups. It does this by calculating two significant values, an **F-Statistic** value and a **p-value**. The F-statistic describes two different variances, a **between-group variance**: how the means differ from each other between the optimizers, and a **within-group variance**: how the results may fluctuate among the optimizer groups they are in. This is calculated with:

$$\square = \frac{\square\square\square\square\square\square\square - \square\square\square\square\square\square\ \square\square\square\square\square\square\square\square\square}{\square\square\square h\square\square - \square\square\square\square\square\square\ \square\square\square\square\square\square\square\square\square}$$

Simultaneously, the p-value is calculated, which measures the ‘significant’ aspect of the test. A **threshold** is stated, which in our case is 0.05, denoting that there is less than a 5% chance that the variance observed occurred due to **randomness**. Thus, after calculation, if the p-value results in a value less than 0.05, it means that it has a significant effect on the performance, as it has less than 5% possibility that it occurred due to randomness. The formula for calculating the p-value for specifically ANOVA tests is:

$$p - value = P(F > F_{obs} \mid df_1, df_2)$$

Where the $F_{obs}$ is the observed F value, $df_1$ represents the **degrees of freedom** for between groups, and , $df_2$ represents the degrees of freedom for within-groups **[14]**.

The formula for both degrees of freedom calculations are:

$$df_1 = k - 1$$

$$df_2 = N - k$$

Where *k* represents the number of groups , which in this case is the number of optimizers. *N* represents how many samples there are in total, which is the number of optimizers multiplied with the number of seeds, which in this case would be 4 optimizers and 4 seeds, thus resulting in *N* taking the value of 16.

With this, two hypotheses will be stated, a **Null Hypothesis** and the **Alternative Hypothesis**. The hypotheses will be formulated for the 4 different performance metrics.

*Null Hypothesis ($H_0$):* There is no statistically significant difference in the average performance metrics across all optimizers. This implies that there is no statistically significant difference between the 4 models with the different optimizers setup, and that any differences observed occurred due to random chance.

*Alternative Hypothesis ($H_1$):* There is at least 1 or more optimizer that results in a statistically significant different value for either of the 4 performance metrics and that the optimizers do act differently from each other.

Below is a table stating the 4 hypotheses formulated for both the Null hypothesis and the Alternative hypothesis. These will be analyzed and later in the results, depending on the outcome of the tests, will either be rejected or accepted.

*Fig. 4, 5: Tables displaying the Null and Alternative hypotheses formulated for the ANOVA statistical testing for the 4 different performance metrics,*

| | **Accuracy** | **Precision** | **F1** | **Recall** |
|---|---|---|---|---|
| **Null Hypothesis** $H_0$ | The accuracy is equal for all optimizers. | The Precision is equal for all optimizers. | The F1 score is equal for all optimizers. | The Recall is equal for all optimizers. |
| **Alternative Hypothesis** $H_1$ | At least one optimizer has a statistically significant difference Accuracy from the others. | At least one optimizer has a statistically significant difference Precision from the others | At least one optimizer has a statistically significant difference F1 score from the others | At least one optimizer has a statistically significant difference Recall value from the others |

### B. Experimental Setup

As mentioned in the architecture section, the controlled experiments were conducted on the four different combinations of optimizers and update strategies across four different random seeds:

**SGD without optimizer** — standard gradient descent with one sample update per iteration

**SGD with AdaM** — adaptive optimizer using per-parameter updates

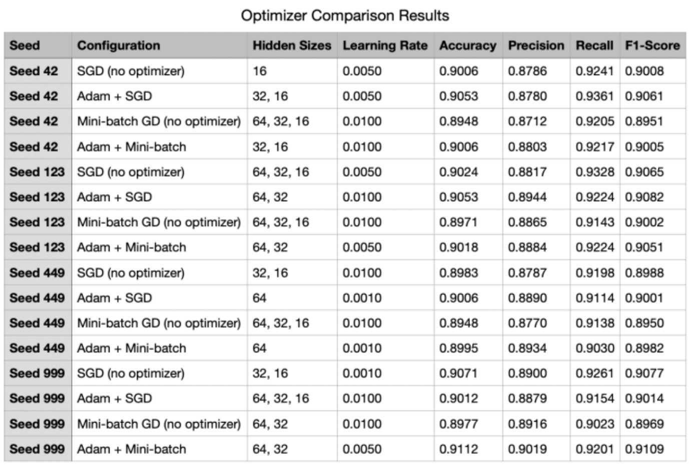
**Mini-batch GD without optimizer** — batch updates without AdaM

**Mini-batch with AdaM** — AdaM optimizer with batch updates

They were run with the same dataset and model, over four arbitrarily chosen seeds (42, 123, 449, 999) to control for variance. Every run used grid-search to tune the hyperparameters, testing 18 combinations of learning rates and hidden layer sizes, and used validation accuracy to select the best performing model. The model was then evaluated once on the test set and the metrics were logged.

## V. Results

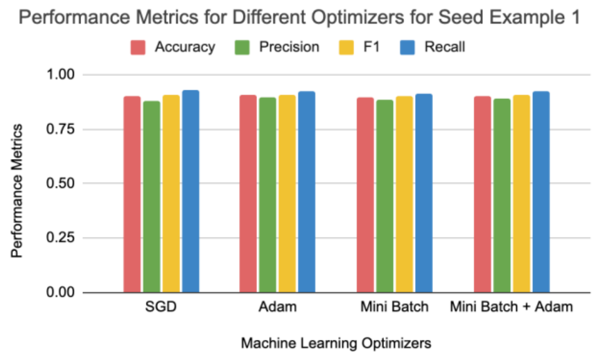
*Fig. 6: Raw results table displaying the optimizer setup comparison results between seed 42, 123, 449, and 999.*

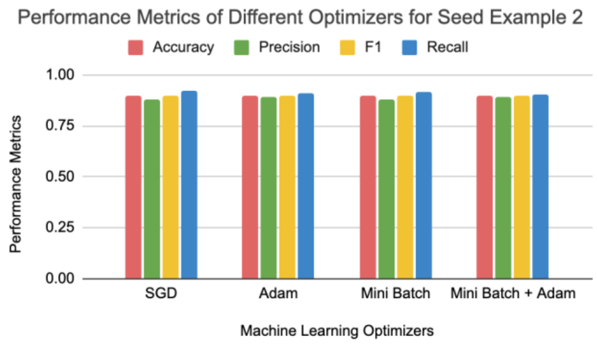


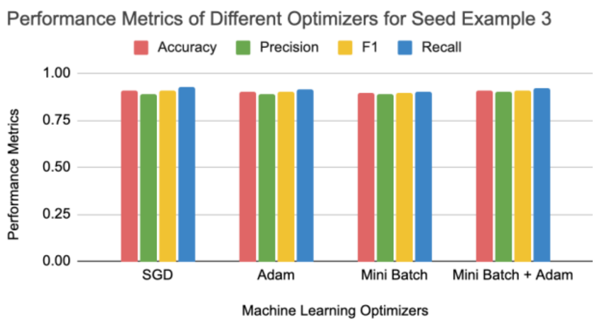
Optimizer Comparison Results

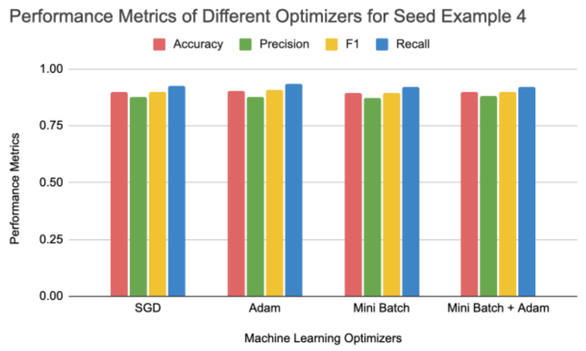
| Seed | Configuration | Hidden Sizes | Learning Rate | Accuracy | Precision | Recall | F1-Score |
|---|---|---|---|---|---|---|---|
| **Seed 42** | SGD (no optimizer) | 16 | 0.0050 | 0.9006 | 0.8786 | 0.9241 | 0.9008 |
| **Seed 42** | Adam + SGD | 32, 16 | 0.0050 | 0.9053 | 0.8780 | 0.9361 | 0.9061 |
| **Seed 42** | Mini-batch GD (no optimizer) | 64, 32, 16 | 0.0100 | 0.8948 | 0.8712 | 0.9205 | 0.8951 |
| **Seed 42** | Adam + Mini-batch | 32, 16 | 0.0100 | 0.9006 | 0.8803 | 0.9217 | 0.9005 |
| **Seed 123** | SGD (no optimizer) | 64, 32, 16 | 0.0050 | 0.9024 | 0.8817 | 0.9328 | 0.9065 |
| **Seed 123** | Adam + SGD | 64, 32 | 0.0100 | 0.9053 | 0.8944 | 0.9224 | 0.9082 |
| **Seed 123** | Mini-batch GD (no optimizer) | 64, 32, 16 | 0.0100 | 0.8971 | 0.8865 | 0.9143 | 0.9002 |
| **Seed 123** | Adam + Mini-batch | 64, 32 | 0.0050 | 0.9018 | 0.8884 | 0.9224 | 0.9051 |
| **Seed 449** | SGD (no optimizer) | 32, 16 | 0.0100 | 0.8983 | 0.8787 | 0.9198 | 0.8988 |
| **Seed 449** | Adam + SGD | 64 | 0.0010 | 0.9006 | 0.8890 | 0.9114 | 0.9001 |
| **Seed 449** | Mini-batch GD (no optimizer) | 64, 32, 16 | 0.0100 | 0.8948 | 0.8770 | 0.9138 | 0.8950 |
| **Seed 449** | Adam + Mini-batch | 64 | 0.0010 | 0.8995 | 0.8934 | 0.9030 | 0.8982 |
| **Seed 999** | SGD (no optimizer) | 32, 16 | 0.0010 | 0.9071 | 0.8900 | 0.9261 | 0.9077 |
| **Seed 999** | Adam + SGD | 64, 32, 16 | 0.0100 | 0.9012 | 0.8879 | 0.9154 | 0.9014 |
| **Seed 999** | Mini-batch GD (no optimizer) | 64, 32 | 0.0100 | 0.8977 | 0.8916 | 0.9023 | 0.8969 |
| **Seed 999** | Adam + Mini-batch | 64, 32 | 0.0050 | 0.9112 | 0.9019 | 0.9201 | 0.9109 |

*Figs. 7, 8, 9, 10: Bar charts displaying the performance metrics for the 4 different optimizer setups for the seed examples 123, 449, 999, and 42.*

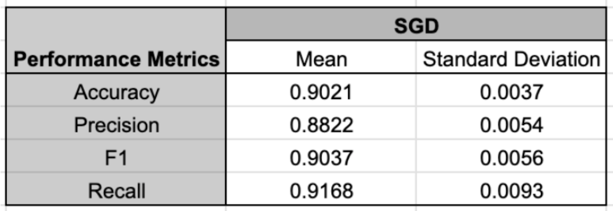




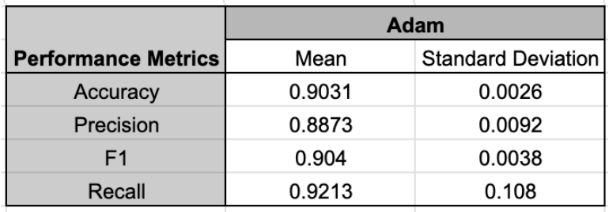




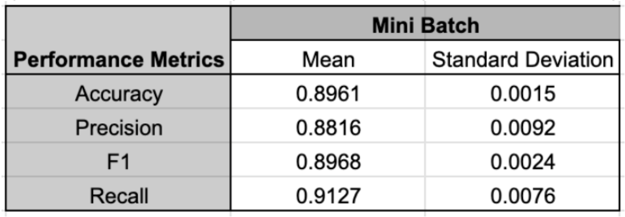
*Figs. 11, 12, 13: Tables displaying the performance metrics for the 4 different optimizer setups with the combined results from the 4 different seed examples*



| | **SGD** | |
|---|---|---|
| **Performance Metrics** | Mean | Standard Deviation |
| Accuracy | 0.9021 | 0.0037 |
| Precision | 0.8822 | 0.0054 |
| F1 | 0.9037 | 0.0056 |
| Recall | 0.9168 | 0.0093 |

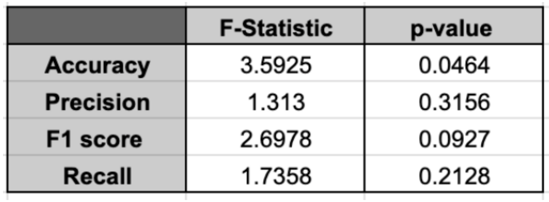


| | **Adam** | |
|---|---|---|
| **Performance Metrics** | Mean | Standard Deviation |
| Accuracy | 0.9031 | 0.0026 |
| Precision | 0.8873 | 0.0092 |
| F1 | 0.904 | 0.0038 |
| Recall | 0.9213 | 0.108 |



| | **Mini Batch** | |
|---|---|---|
| **Performance Metrics** | Mean | Standard Deviation |
| Accuracy | 0.8961 | 0.0015 |
| Precision | 0.8816 | 0.0092 |
| F1 | 0.8968 | 0.0024 |
| Recall | 0.9127 | 0.0076 |

*Fig. 14: Table displaying the statistical testing calculated for the performance metrics between the 4 different optimizers setups from the 4 different seed examples.*



| | F-Statistic | p-value |
|---|---|---|
| Accuracy | 3.5925 | 0.0464 |
| Precision | 1.313 | 0.3156 |
| F1 score | 2.6978 | 0.0927 |
| Recall | 1.7358 | 0.2128 |

## VI. Evaluation

The study explored and tested 4 different models built with four different optimization strategies, stochastic gradient descent (SGD), SGD with AdaM, mini-batch gradient descent, mini-batch with AdaM; in order to analyze its performance for predicting diabetes. The performance was measured by analyzing 4 different performance metrics, accuracy, precision, F1-score and recall. After processing raw results, statistical analysis was performed to observe insights into the results and what they mean.

Taking a look at the mean and the standard deviation calculated in Figures 9, 10, 11, and 12, the differences between mean for all the performance metrics for every optimizer are minor and subtle, which applies to the standard deviation as well, revealing consistency. The mean result was the highest for the mini-batch + AdaM with the performance metric accuracy resulting with a 0.9033, however having the lowest standard deviation, indicating that mini-batch + AdaM remained the most consistent.

Moving further to the ANOVA tests, the differences were better measured and calculated. Looking at Figure 13, the only p-value that was less than 0.5, was for the performance metric **Accuracy**, which had a **p-value of 0.0464**, whereas the **other metrics** had **p-values higher than 0.9** up to **0.3**. This value supports the hypothesis mentioned earlier how at least one of the optimizers, which in this case is Accuracy, has a statistically significant impact on the performance of the optimizers predictive behavior. The remaining 3 metrics, precision, recall and F1-score, displayed no statistically significant difference with their p-values, that suggest that these optimizers may perform well by influencing how correct and accurate the predictions are, however do not affect the way the optimizer behaves regarding the individual classes and mostly focuses on how well the model handles false negatives,true positives and false positives.

Exploring this further, the reasoning behind why this may occur, is due to the nature of each optimizer and how weights are handled. When looking at SGD, after every individual sample, the weights are updated, allowing for a better responsiveness. However, this could lead to volatility as well as the occurrence of frequent fluctuations, which could explain the standard deviation values being high for SGD in figure 9. The AdaM optimizer differs by implementing momentum and adaptive learning rates, meaning that weights of each individual are updated based on the previous gradients. This can be observed in figure 10 with the standard deviation of 0.0021 as one of the smallest standard deviation values, displaying **stability**. This is beneficial for noisy datasets as it allows stability **[5]**. The mini-batch gradient descent also promotes stability as with the splitting into mini-batches, the averages of each small batch reduces the risk and interferences of outliers in the samples. This is observed in table 11 with an F1 score of 0.8968, suggesting that it shows a moderate sense of stability when measuring between the 4 trials. Now combining AdaM with mini-batch, the performance is amplified, as mentioned earlier, by having the highest accuracy and the highest F1-score when observing the mean. This implies that this optimizer had excellent consistency, thus supporting the choice of developing models with a combination of both, the process of smoothing batches and having **adaptive** updates **[4]**.

As a result of the performance of the mini-batch and AdaM combined, it offers stability when training with medical datasets that are specifically **tabular**. Thus, as this was done on a simpler dataset compared to more complex and imbalanced datasets, ideas for further work will be proposed.

To further expand this project, the models can be tested and evaluated on actual datasets that are significant and used in medical environments such as the Electronic Health Record, where imbalance and noise is prevalent due to the disordered nature of authentic human medical data, that allows testing to be done in high dimensional and intricate environment. The hypothesis to be formulated for this investigation would be to hypothesize and see whether the optimizers that combine mini-batch and the AdaM optimizer maintain the efficient performance when measuring the accuracy and comparing it to other models **[12]**.

Looking more technically, a worthy experiment is to see how the intricacy of the neural networks interact with the optimizers. This directly focuses on the hidden layers, with varying the width as well as the depth of the architecture, understanding their relationship and observing correlations and causations. This would give insight to both the vanishing and exploding gradient situation, as due to the method of propagation backwards through multiple layers, how they result in being either too small or too large, causing instability **[13]**.

Thus regarding the confidence of the claims made throughout this report, hold strength due to the results of the intensive statistical tests done and observing consistency across all 4 seeds. This heavily supports the claim that the choice and development of optimization methods significantly affects the behavior of the model, specifically when looking at the accuracy of prediction. The analysis also gave insights to which performance metrics do not contribute to enhancing the design and performance of the optimizer.

## VII. Conclusion

This study explored the implementation of a custom Multi-Layer Perceptron model, developed from scratch using only the NumPy library - to classify diabetes based on patient data. The main focus was placed on evaluating the impact of different optimization strategies, namely stochastic gradient descent, mini-batch gradient descent, and the combinations of these with the AdaM optimizer; on model performance. Among the strategies tested, combining mini-batch gradient descent with the AdaM optimizer delivered the most consistent performance based on the highest average accuracy and lowest standard deviation. This consistency is quite valuable in medical applications, where reliability is very important.

While *accuracy* **did** have **significant** differences across the optimizers, the **other metrics** like *precision*, *recall*, and *F1-score* **did not**. This indicates that the optimizer selection may affect the model's correctness, but not necessarily how it handles individual classes.

This study also examined feature importance to assess the relative contribution of different input variables. The *gender* feature was found to have the least impact on diabetes prediction, suggesting that possibly other physiological and lifestyle-related features play a bigger role in determining the possibility of diabetes.

This research demonstrates the potential of neural networks in medical diagnosis when properly optimized. While machine learning models should not replace human expertise, they can serve as decision-support tools, helping medical staff in detection and risk assessment. Future work could extend this study by incorporating additional feature selection techniques, possibly experimenting with deeper architectures, or applying more advanced optimization methods to enhance predictive accuracy and generalizability.

## References

**[1]** T. Panch, P. Szolovits, and R. Atun, "Artificial intelligence, machine learning and health systems," *Journal of Global Health*, vol. 8, no. 2, Oct. 2018, doi: https://doi.org/10.7189/jogh.08.020303.

**[2]** S. Secinaro, D. Calandra, A. Secinaro, V. Muthurangu, and P. Biancone, "The role of artificial intelligence in healthcare: a structured literature review," *BMC Medical Informatics and Decision Making*, vol. 21, no. 1, Apr. 2021, Available: https://link.springer.com/article/10.1186/s12911-021-01488-9**.**

**[3]** N. Shahid, T. Rappon, and W. Berta, "Applications of artificial neural networks in health care organizational decision-making: A scoping review," PLOS ONE, vol. 14, no. 2, p. e0212356, Feb. 2019, doi: https://doi.org/10.1371/journal.pone.0212356.

**[4]** I. Goodfellow, Y. Bengio, and A. Courville, Deep Learning. MIT Press, 2016.

**[5]** D. Kingma and J. Lei Ba, "ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION," 2015. Available: https://arxiv.org/pdf/1412.6980.

**[6]** L. Datta, "A Survey on Activation Functions and their relation with Xavier and He Normal Initialization," arXiv (Cornell University), Mar. 2020, doi: https://doi.org/10.48550/arxiv.2004.06632.

**[7]** D. Martin and Ailab, "Evaluation: From precision, recall and F-measure to ROC, informedness, markedness & correlation," *ResearchGate*, 2011. https://www.researchgate.net/publication/276412348_Evaluation_From_precision_recall_and_F-measure_to_ROC_informedness_markedness_correlation

**[8]** R. Shen, L. Gao, and Y.-A. Ma, "On Optimal Early Stopping: Over-informative versus Under-informative Parametrization," *arXiv.org*, 2022. https://arxiv.org/abs/2202.09885.

**[9]** F. K. Ewald, L. Bothmann, M. N. Wright, B. Bischl, G. Casalicchio, and G. König, "A Guide to Feature Importance Methods for Scientific Inference," *Communications in Computer and Information Science*, pp. 440–464, 2024, doi: https://doi.org/10.1007/978-3-031-63797-1_22.

**[10]** *The authors mention that the original source of the dataset remains undisclosed to ensure the privacy of the patients that participated in the research. The dataset was acquired from the following link:* https://www.kaggle.com/datasets/iammustafatz/diabetes-prediction-dataset

**[11]** P. Liashchynskyi, “Grid Search, Random Search, Genetic Algorithm: A Big Comparison for NAS,” arXiv:1912.06059 [cs, stat], Dec. 2019, Available: https://arxiv.org/abs/1912.06059

**[12]** S. Wang, B. Lei, Z. Yu, Y. Wang, and T. Feng, "Handling Imbalanced Medical Image Data: A Deep-Learning-Based One-Class Classification Approach," *Artificial Intelligence in Medicine*, vol. 108, p. 101935, 2020. [Online]. Available: https://doi.org/10.1016/j.artmed.2020.101935

**[13]** E. M. Dogo, O. J. Afolabi, and B. Twala, "On the Relative Impact of Optimizers on Convolutional Neural Networks with Varying Depth and Width for Image Classification," *Applied Sciences*, vol. 12, no. 23, p. 11976, Nov. 2022. [Online]. Available: https://doi.org/10.3390/app122311976

**[14]** D. C. Montgomery, *Design and Analysis of Experiments*, 10th ed. Hoboken, NJ, USA: John Wiley & Sons, 2019. [Online]. Available: https://www.wiley.com/en-us/Design+and+Analysis+of+Experiments%2C+10th+Edition-p-9781119492443

**MLVU final report information sheet**

**Group number:** 63

**Authors**

| name | student number |
|---|---|
| M. Fatih A. | 2802560 |
| Kayra Ö. | 2813857 |
| Yusuf A. | 2808418 |
| Tim v.d. T. V. | 2775600 |
| Vaamika R. | 2800454 |

**Software used**
*The algorithm is a Multi-Layer Perceptron with specific design choices, tasked with binary classification. The only library used in the implementation in Python, is NumPy. For the development, VSCode was used as the code editor. For branch and commitment management, this project also utilized a Github repository. Sklearn was used outside of the algorithm itself, to provide results for which we can compare to ours.*

**Use of AI tools**
*ChatGPT and similar transformer models were used throughout the project for brain-storming, bug fixing in the code, getting ideas and implementing solutions. Github Copilot was enabled on one of the developers' computer, however the code is mostly hand-written. (Copilot was used for simple comments or the creation of the docstrings, etc. and the rest of the implementation of the algorithm is hand-written.)*

**Link to code (optional)**
*Git repository: https://github.com/AresPaleson/MLP-from-scratch.git*
*Secondary git repository: https://github.com/thewyveo/MLP-only-numpy.git*