Building a Simple Neural Network and Parameter Tuning the Optimiser

Dean Sayre

This document embodies a general discussion of the mechanisms required for building a simple Neural Network, these include: Forward Propagation, Backward Propagation, and Optimisation. Amongst the mechanisms, optimisation is studied in depth between three optimisers (Gradient Descent, Gradient Descent + Momentum, and ADAM). The discussion of these optimisers includes a simple implementation of hyper-parameter tuning for each. As a result, optimal parameters for each optimiser are located, which prompts a continued study of comparing the competing optimisers. This simple Neural Network is applied to a data set of 178 wines, each belonging to one of three wine classes, defined by 13 wine features. This study will show that of the three optimisers applied on this data, the ADAM optimiser converges faster than the gradient descent based optimisers.

I. INTRODUCTION

Neural Networks (NNs) are a key piece of some of the most successful machine learning algorithms. The development of neural networks have been key to teaching computers to solve problems the way humans do. In theory, a neural network emulates the human brain. Brains cells, or neurons, are connected via synapses. In NNs, this natural phenomena takes the form of nodes (neurons) connected by weighted edges (synapses).

The capabilities of NNs are diverse and have been implemented for a variety of practical problems: Convolutional NNs classify sound/images/and videos, Autoencoders implement dimensionality reduction of its input data, Probabilistic NNs locate patterns and can be used for classification. However, the difficulty of implementation and understanding scales up with the difficulty of the problem. For the sake of introduction to implementations of NNs I share a simple build of a NN, popularly known as an Artificial Neural Network (ANN).

As stated previously, ANNs are composed of a series of neurons/nodes connected together by weighted values. This series will contain an input layer, some number of hidden layers, and an output layer. Fig. 1 shows an example of a generic NN with the following connections: Three nodes in the input layer –> four nodes in hidden layer 1 –> four nodes in hidden layer 2 –> one node in the output layer. The weighted values are shown as black lines connecting nodes in adjacent layers. One can increase the complexity of their ANN by increasing the number of hidden layers. However, the important part is that the inputs and outputs of each neuron are weighted, and if implemented correctly, the ANN should be able to execute what it is programmed for.

As previously mentioned, ANNs have the capability to solve a variety of problems, but these problems in general fall under the categories of regression and classification. In regression problems, the NN aims to map input variables (features) to some continuous function (hypothesis). In practice, this could be fitting a functional form to a set of measured data. In classification problems, the goal of the NN is to teach the model how to assign labels to input data. This consists of mapping input features to discrete categories or classes. Regardless of the problem, the aim of the NN is to train it's weighted connections over many iterations so that it is capable of fitting/classifying the data correctly. In this paper, we are concerned with the problem of classification.

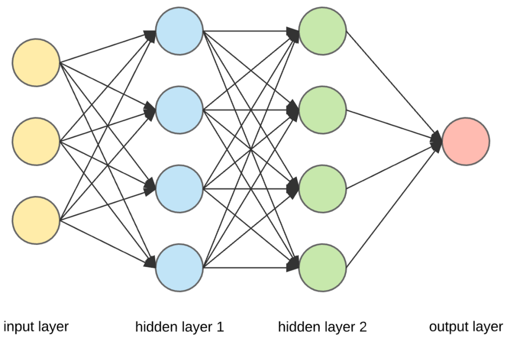


Fig. 1. Example of a NN with two hidden layers (blue and green) with four nodes each. The arrows between nodes are the weights (or weighted connections). This neural network is designed to take 3 inputs and produce a single output, a configuration for a regression problem.

In this paper, an ANN is applied to a classification problem. The network will have the capability to take in some data set and, with high accuracy, correctly classify them to their target categories. The model's success is studied through three different NN optimisers. Introduced more in depth later, these optimisers are Gradient Descent, Gradient Descent + Momentum, and ADAM. The comparison will include implementation speed, and speed of convergence (more on this later). The remainder of the paper will be structured in the following way: (II) Introduction of the data set, (III) discussion of the structure of the NN, (IV) implementation of parameter tuning, (V) a model comparison, and (VI) concluding statements.

II. DATA SET

The data set is composed of 178 samples of wine, each belonging to 3 categories, described by 13 features (Alcohol, Malic acid, Ash, Alcalinity of ash, Magnesium, Total phenols, Flavanoids, Nonflavanoid phenols, Proanthocyanins, Color intensity, Hue, OD280/OD315 of diluted wines, and Proline). These features are the results of a chemical analysis of wines grown in the same region in Italy but derived from three different cultivars. The analysis determined the quantities of these 13 features found in each of the three types of wines. To obtain the data, one simply needs to load the data set in the following manner:

```
from sklearn.datasets import load_wine

data=load_wine()
x,y = data.data, data.target
```

Loading the data from scikit-learn allows one to conveniently split the data into features (x) and target categories (y). The features are denoted by floating-point numbers, while the categories are denoted by 0, 1, and 2. Fig. 2 shows the data. Prior to training any data set in a NN, it is important to normalize the data, particularly its features. This is done by simply changing the values of the data to a common scale. This is a vital step in the event that features have different ranges. Normalization can be done with ease by using scikit-learn's standard scaling module:

```
from sklearn.preprocessing
        \ import StandardScaler

scaler = StandardScaler()
scaler.fit(x)
x0 = scaler.transform(x)
```

Scikit-learn's scaling module uses Eq. (1)

$$x0 = \frac{x - u}{s} \tag{1}$$

Where x is the input feature, and u and s are the mean and the standard deviation of the training samples, respectively. The distribution of unscaled (x) and scaled values (x0) are shown in Fig. 3. Notice that the range of normalized values decreases by about 3 orders of magnitude in comparison to the unnormalized.

Although we focus on one data set, scikit-learn is home to a large database, all in which can be used in either classification or regression problems. Although it may not be necessary to scale all data sets, it is an important practice that will contribute to efficient success of the NN. In the next section, I break down the inner structure of neural networks.

III. NEURAL NETWORK STRUCTURE

In general, neural nets are structured in the following way: (A) Forward Propagation, (B) Backward Propagation, and (C) Optimization. The aforementioned steps are repeated until convergence of the NN or until a set amount of iterations is reached.

A. Forward Propagation

The forward propagation step of a NN is visualized in Fig. 1. The weighted connections between the input layers and the hidden layers, and the operations performed within each node of the hidden layers, eventually lead to some output layer. In this last layer, the model has made its prediction. Thus, a forward propagation, amongst other tasks, will always lead to a prediction in the output layer. For the sake of wine prediction, the model is organized in the following manner:

First Layer/Input Layer: This layer will comprise 13 nodes, each node representing a normalized input feature obtained from scikit-learn's wine data set.

Hidden Layer 1: This layer will have 4 nodes, or 4 hidden nodes. This layer is connected to the input layer via the weighted values. Since there are 13 nodes in the Input Layer, and 4 in this layer, there are a total of 52 weighted connections between the two layers. Within each hidden layer node lies an activation function, used to include non-linearity to the model. The calculations between the Input Layer and Hidden Layer 1 can be described by Eq. (2) and Eq. (3):

$$z_1 = w_1 \cdot x_0 + b_1 \tag{2}$$
$$h_1 = \sigma_h(z_1) \tag{3}$$

Where x_0 is the input data, w_1 and b_1 are the weighted connections between the Input Layer and Hidden Layer 1 and some bias, respectively, z_1 is the first Forward Prop. step, σ_h is some activation function, and h_1 is the activated Forward Prop. step.

Hidden Layer 2: Connected to hidden layer 1, hidden layer 2 will also have 4 "activated" hidden nodes. This layer takes in the output of hidden layer 1, connected by weighted outputs. In total, there are 16 weighted connections between the first and second hidden layers. There are similar equations as Eq. (2) and Eq. (3) describing the connection between the two hidden layers, given by:

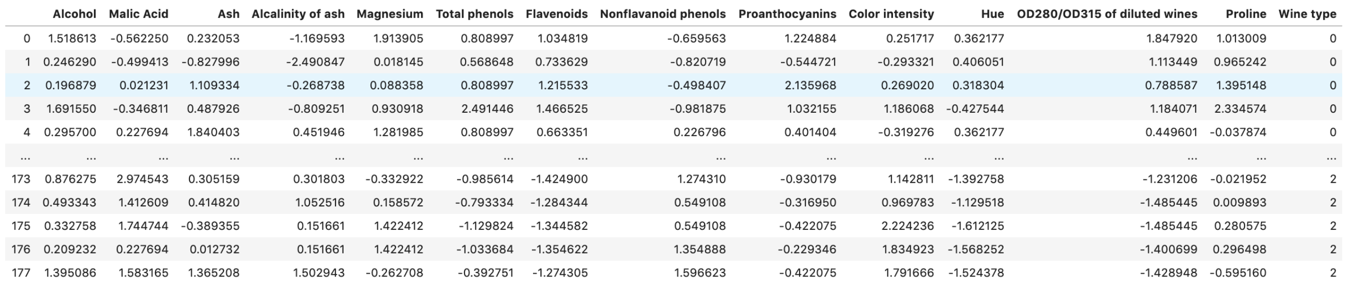
$$z_2 = w_2 \cdot h_1 + b_2 \tag{4}$$
$$h_2 = \sigma_h(z_2) \tag{5}$$

The parameters in Eq. 4 and 5 have similar purpose to the equations describing the previous layer connection.

Output Layer: The final layer of the model acts as the prediction of the neural network. This will take the form of three nodes, each corresponding to the target wine class (0, 1, or 2). Because of the choice of activation function in this layer, the sum of nodal outputs will always be 1. Thus each nodal output represents the probability that the prediction falls within either class. The Output Layer is connected to the Second Hidden Layer by the following formulae:

$$z_3 = w_3 \cdot h_2 + b_3 \tag{6}$$
$$y\_ = \sigma_o(z_3) \tag{7}$$

Again, these Eqs. 6 and 7, are similar to those previously presented. Of most importance here is y_, which represents the NNs predicted classification.



	Alcohol	Malic Acid	Ash	Alcalinity of ash	Magnesium	Total phenols	Flavenoids	Nonflavanoid phenols	Proanthocyanins	Color intensity	Hue	OD280/OD315 of diluted wines	Proline	Wine type
0	1.518613	-0.562250	0.232053	-1.169593	1.913905	0.808997	1.034819	-0.659563	1.224884	0.251717	0.362177	1.847920	1.013009	0
1	0.246290	-0.499413	-0.827996	-2.490847	0.018145	0.568648	0.733629	-0.820719	-0.544721	-0.293321	0.406051	1.113449	0.965242	0
2	0.196879	0.021231	1.109334	-0.268738	0.088358	0.808997	1.215533	-0.498407	2.135968	0.269020	0.318304	0.788587	1.395148	0
3	1.691550	-0.346811	0.487926	-0.809251	0.930918	2.491446	1.466525	-0.981875	1.032155	1.186068	-0.427544	1.184071	2.334574	0
4	0.295700	0.227694	1.840403	0.451946	1.281985	0.808997	0.663351	0.226796	0.401404	-0.319276	0.362177	0.449601	-0.037874	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
173	0.876275	2.974543	0.305159	0.301803	-0.332922	-0.985614	-1.424900	1.274310	-0.930179	1.142811	-1.392758	-1.231206	-0.021952	2
174	0.493343	1.412609	0.414820	1.052516	0.158572	-0.793334	-1.284344	0.549108	-0.316950	0.969783	-1.129518	-1.485445	0.009893	2
175	0.332758	1.744744	-0.389355	0.151661	1.422412	-1.129824	-1.344582	0.549108	-0.422075	2.224236	-1.612125	-1.485445	0.280575	2
176	0.209232	0.227694	0.012732	0.151661	1.422412	-1.033684	-1.354622	1.354888	-0.229346	1.834923	-1.568252	-1.400699	0.296498	2
177	1.395086	1.583165	1.365208	1.502943	-0.262708	-0.392751	-1.274305	1.596623	-0.422075	1.791666	-1.524378	-1.428948	-0.595160	2

Fig. 2. Table of normalized data. For each wine sample there are 13 characteristic features. The last column give the true wine type (i.e. label).

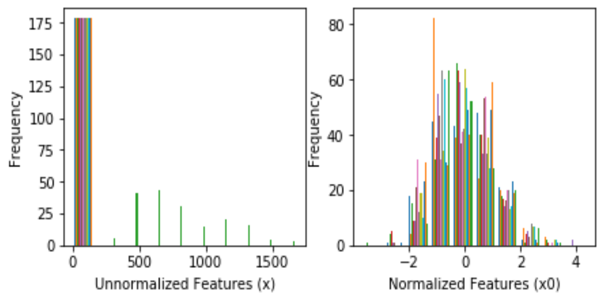


Fig. 3. Distribution of unnormalized (right) and normalized (left) wine dataset features.

Activation Functions: As mentioned earlier, activation functions introduce some non-linearity to the NN. Without the introduction of non-linearity, each layer output will be linear, and it would be as if the stacking of multiple hidden layers was simply a single linear layer. This reduces the complexity of the NN, possibly limiting its capabilities. In this paper, we consider the Sigoid activation function (Eq. 8) for Hidden Layer 1 and 2 and the Tanh activation function (Eq. 9) for the output layer. Choice of these activation functions ensures that the outputs are probabilities of a prediction belonging to either wine class.

$$\sigma_h = \frac{1}{1+e^{-z}} \tag{8}$$

$$\sigma_o = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{9}$$

Cost Function: How does one know how well the model is doing? Well, one could simply wait until all training has been completed and run the same training, or some separate test, data through the model's forward pipeline. If the NN predicts all input data correctly with a high percentage, the model worked! But what if one would like to track the model's progress every step? This can be done through the cost function. For this study, we use the Logarithmic Loss function Eq. (10),

$$L = \frac{1}{m}\sum_{i\in C} y_i \log y\_i \tag{10}$$

which measures the uncertainty of the probabilities of the model by comparing them to the true labels. Most importantly, Eq. (10) will be used during the back propagation step. Here, $y\_$ and y (true wine class) are compared over C wine categories. The underlying goal of the neural network is to decrease the loss until convergence, which is performed in the next two subsections. Convergence here is defined as when $L_i \approx L_{i-1}$, were i represents some iteration.

B. Backward Propagation

The goal of back propagation in NNs is arguably the most vital. Since the NN is dependent on its model parameters, its weights and biases, to make correct predictions, the model needs a way to change the parameters in case a bad prediction was made (formally stated, in the event that the cost function is high). This begins in the back propagation step, where we are interested in the gradient of the loss w.r.t all model parameters $(w_1, b_1, w_2, b_2, w_3, b_3)$. If the mathematics is correct, the gradients will point in the direction that iteratively decreases Eq. (10). The parameter updates using the gradients is explained in the next section.

C. Optimization

Upon obtaining the gradient of L w.r.t all parameters $(w1, w2, w3, b1, b2, b3)$, the NN needs a scheme to update the weights for the next iteration. This is done through optimization. Here, three optimization techniques are used.

Gradient Descent: Of the two techniques, Gradient Descent (GD) is the easiest to apply in practice. Upon obtaining all necessary gradients, GD updates each parameter using Eq. (11) and Eq. (12):

$$w_i = w_{i-1} - \eta * \beta_1^i * \frac{dL}{dw_{i-1}} \tag{11}$$

$$b_i = b_{i-1} - \eta * \beta_i^i * \frac{dL}{dw_{i-1}} \tag{12}$$

where w_i denotes the weights for the current iteration, w_{i-1} denotes the weights for the previous iteration, η

is the learning rate, β_1 is the decay rate for the current iteration, and $\frac{dL}{dw_{i-1}}$ is the gradient of the loss (Eq. (10)) w.r.t the weighted connections. Eq. (11) and Eq. (12) assumes that w and b are vectors containing all-layer weights and biases, respectively.

Of most interest is the hyper-parameter η, which controls the amount that the parameters w and b are updated in the NN model. Formally known as the learning rate, η controls the speed at which the model learns. When the learning rate is too large, gradient descent can inadvertently increase rather than decrease the training error. When the learning rate is too small, training is not only slower, but may become permanently stuck with a high training error. Therefore, we should not use a learning rate that is too large or too small. Optimal values for this hyper-parameter lie between 0.0 and 1.0. Later, parameter tuning will be implemented to find the optimal value for the current problem.

Gradient Descent + Momentum: The GD algorithm can be expanded by adding a momentum term (v) to the previous algorithm. The intuition behind momentum is to continue updating the parameter along the previous update direction. This algorithm is given by

$$v_i = \mu * v_{i-1} + (1-\mu) * \frac{dw_{i-1}}{dL} \tag{13}$$

where v is the momentum term, and μ is the momentum parameter that typically lies between 0 and 1. A weight/bias update can then be implemented with 14

$$w_i = \eta * v_i - w_{i-1} \tag{14}$$

The update for the bias parameter is identical to 14.

ADAM: Another optimization algorithm that can be used to update network weights iteratively is ADAM, whose name is derived from Adaptive Moment Estimation. ADAM is the combination of two successful optimization techniques Adagrad and RMSProp. The main difference between GD and ADAM is that ADAM computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients (Eq. (15) and Eq. (16)), whereas the learning rates used in GD are iteratively the same for all parameter. It is important to be aware that the ADAM optimization technique is an approximate second-order optimizer.

$$(\frac{\partial w_i}{\partial L})_{mean} = \beta_1 * (\frac{\partial w_{i-1}}{\partial L})_{mean} + (1-\beta_1) * \frac{\partial w_{i-1}}{\partial L} \tag{15}$$

$$(\frac{\partial w_i}{\partial L})_{variance} = \beta_2 * (\frac{\partial w_{i-1}}{\partial L})_{variance} + (1-\beta_2) * (\frac{\partial w_{i-1}}{\partial L})^2 \tag{16}$$

From the above equations, β_1 and β_2 are the first and second decay rate. Note that the first and second moments for the gradients of the biases have a similar form to that of (Eq. (15) and Eq. (16)).

After obtaining the first and second moments of the gradient, updates are in the simple form of 17:

$$w_i = -w_{i-1} + \eta * \frac{(\frac{\partial w_i}{\partial L})_{variance}}{\sqrt{(\frac{\partial w_i}{\partial L})_{mean} + \epsilon}} \tag{17}$$

As in GD η is the learning rate, and ϵ is some factor to prevent the update of η from diverging to infinity. Again, the update for the model's biases is performed in the same way.

Now that the reader is aware of the basic structure of NNs, he/she is in a good place to implement the model. In the discussion to follow, a comparison between the three optimizers, used to control the adjustments of weights in the NN, will be employed. The comparison will include implementation speed, and speed of convergence. Again, convergence is defined as the model being able to successfully classify some similar test data set of wines.

IV. PARAMETER TUNING

Optimization is a vital step in neural networks to iteratively update model parameters. Eventually, the model must find parameters that force the NN cost to converge. Once this happens, the model (in theory) should have the capability to take in a similar data set that it hasn't seen, and categorize the data correctly. We begin studying each optimizer by parameter tuning its hyper-parameters.

Gradient Descent has two hyper-parameters β_1 and η. Firstly, a decay rate $\beta_1 = 0.9$ is chosen, which simply controls how the learning rate η decays over time. The smaller that η becomes for higher iterations, the less likely that the minimum of the gradient of the weights w.r.t the loss is "jumped" over. However, too small of an η can increase convergence time dramatically. While β_1 remains constant, four values of η are tested: 0.1, 0.01, 0.001, 0.0001. To determine which learning rate is the best for the NN, refer to Fig. 4.

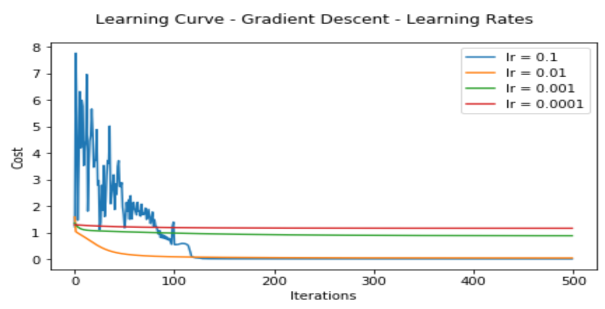


Fig. 4. Cost as a function of iterations for four NN simulations. Each line corresponds to the training curve using gradient descent, with varying η.

From Fig. 4, one can quickly conclude that learning rates of 0.001 and 0.0001 fail to converge to a value that truly minimizes the cost in a timely manner, while learning rates of 0.1

and 0.01 succeed. Notice however, that the orange line (line corresponding to a learning rate of 0.01) converges faster than that of 0.1. Additionally, notice the oscillatory behavior of the blue line. Although there is convergence, a learning rate of 0.1 takes longer to converge due to the observable noise. Although this could probably be fixed by some choosing β_1 to be bigger, for simplicity a learning rate of 0.01 is chosen for GD.

Gradient Descent + Momentum is closely related to gradient descent. This optimizer has two tunable parameters: η and μ (the learning rate and the momentum parameter). Due to the close relation to vanilla GD, η = 0.01 is chosen here while μ is studied with different values: 0.1, 0.3, 0.6, 0.9, 0.99. A plot (Fig. 5.) of the learning curve for this test is shown below.

According to the figure, μ = 0.1, 0.3, and 0.6 converge relatively quickly to a suitable cost value, while μ = 0.90 takes about 500 iterations to converge to the same point. Notice also that, although μ = 0.99 seems to still be decreasing, the fact that it hasn't came close to converging within 500 iterations is reason enough to get rid of it. Since it appears that μ = 0.1 converges the quickest, this is the value chosen for GD + Momentum.

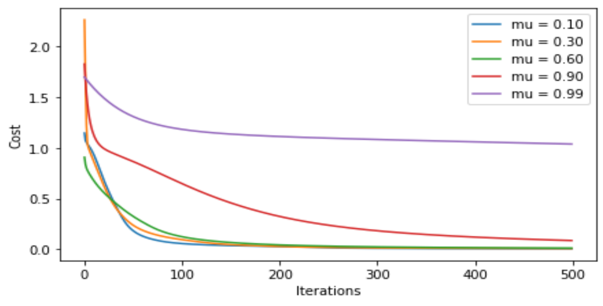


Fig. 5. Cost as a function of iterations for four NN simulations. Each line corresponds to the training curve using gradient descent + momentum, with varying μ and fixed η.

ADAM, compared to the previous two optimisers, is the most involved. It consists of four tunable parameters in total. Three of the four, $\beta_1, \beta_2, \epsilon$ with the values of 0.9, 0.999, $1*10^{-8}$, have been claimed as optimal. Thus, for ADAM, the only hyper-parameter available for tuning is η. Here, η = 0.1, 0.01, 0.001, and 0.0001 is tested. Again, this comparison can be seen in a figure of the learning curve, Fig. 6. Dissimilar to previous results, Fig. 6 shows that the largest learning rate leads to a faster convergence. Thus, we throw out all but η = 0.1.

The optimal parameters to use for each optimizer in the NN have now been determined. For Gradient Descent, it was found that $\beta_1 = 0.9$ and $\eta = 0.01$ lead to the fastest convergence. Using Gradient Descent + Momentum, $\eta = 0.01$ and $\mu = 0.1$, did the best job. Finally, for ADAM, $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 1 * 10^{-8}$, yielded the best results. In the discussion to follow is a comparison of the performance of these three models.

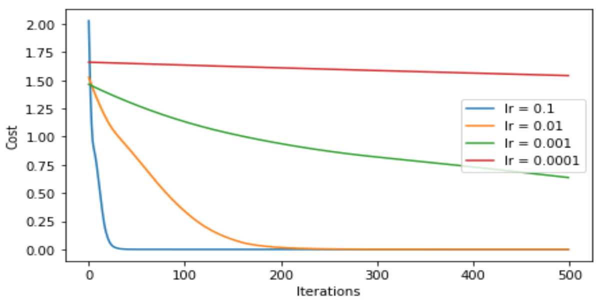


Fig. 6. Cost as a function of iterations for four NN simulations. Each line corresponds to the training curve using ADAM, with varying η and fixed $\beta_1, \beta_2, \epsilon$.

V. MODEL COMPARISON

The performance of the NN in three instances of varying optimisers, using the hyper-parameters chosen in the previous section, is to follow. A maximum of 300 iterations is ran for each model. The time to complete these 300 iterations is compared. The cost is monitored every iteration. Further, if the model reaches a tolerance of $|L_{i-1} - L_i| < 1*10^{-3}$, convergence is raised and the current iteration is saved for comparison. Below is the python script used to make the comparisons:

```
NN_GD = ANN(hidden=[4,4], max_iter=300 ,
    \ optimizer = 'GD',learning_rate=0.01,
    \ decay_rate1=0.99, mu = None,
    \ decay_rate2=None,epsilon=None)

NN_GDM = ANN(hidden=[4,4], max_iter=300 ,
    \ optimizer ='GD+M',learning_rate=0.01,
    \ decay_rate1=None, decay_rate2=None,
    \ mu = 0.10,epsilon=None)

NN_ADM = ANN(hidden=[4,4],max_iter=300,
    \ optimizer='ADAM',learning_rate=0.1,
    \ decay_rate1=0.9, decay_rate2=0.99,
    \ mu = None,epsilon=1e-8)

%timeit -n 10 -r 10 NN_GD.train(x0,y0)

%timeit -n 10 -r 10 NN_GDM.train(x0,y0)

%timeit -n 10 -r 10 NN_ADM.train(x0,y0)
```

In lines 346 - 359, the NN parameters are defined. These include the number of hidden layers/number of neurons per hidden layer, number of iterations, optimizer type, optimizer learning rate, optimizer decay rate 1 2, optimizer momentum, and optimizer epsilon. The following lines, 361-365 initiate the NN training over input wine data x0 and y0. The %timeit decorator times the execution of a Python statement or expression. With timeit, each model is ran 10 times (-r 10) with each run containing 10 loops (-n 10). The mean of each run, obtained from the results of the 10 loops, is calculated, and the best mean + std.deviation from all 10 runs is printed. The simulation results are shown below in Table V, column 2.

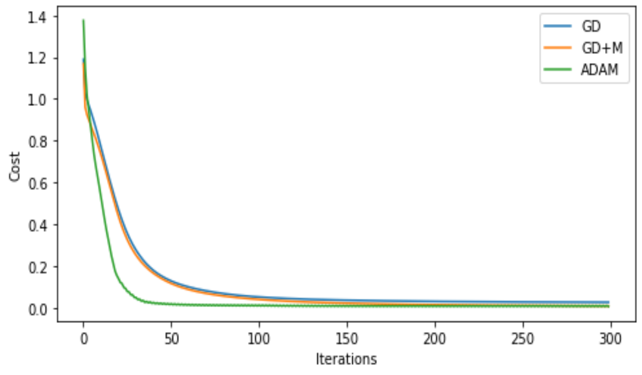


Fig. 7. Learning curve for three different optimizers with chosen hyperparameters.

Optimizer	Time (300 Iter.)	Iter. Converged
GD	52.5 ms ± 1.14 ms per loop	84
GD + M	54.6 ms ± 851 μs per loop	97
ADAM	58.6 ms ± 748 μs per loop	29

Table V shows a model comparison in time and iteration convergence for three NN optimisers.

The time comparisons in Table V column 2 show that the GD optimisers are faster than ADAM, but by only 2-6 ms. However, by observing Fig. 7, which shows the learning curve for the three different optimisers, one can identify that the model ADAM (green line) converges faster (iteration 29) than the gradient descent model (iteration 84) and gradient descent + momentum model (iteration 97). These iterations of convergence are also tabulated in Table V column 3. Notice also that the two gradient descent models barely diverge from one another, during the whole training process. Upon running the code block several times, its worth noting that the gradient descent algorithms do not consistently outperform one another (they consistently switch places), but ADAM consistently outperforms them both.

VI. CONCLUSION

In general, there are only a handful of mechanisms that go into NNs. These mechanisms are broken into three categories: Forward Propagation, Backward Propagation, and Optimization. The goal of Forward Propagation is to take in an example from the data, run it through the NN, and produce a prediction. In classification problems, this prediction is an assignment of the example to a particular class or label. If the prediction is wrong (monitored by the cost function), the NN implements Backward Propagation. In Backward Propagation, the gradients of the cost function w.r.t the model weights and biases is obtained, which symbolize how the model parameters should change in order to reduce the NN cost. These changes are implemented in the optimisation step, where the model parameters are updated. These mechanisms are repeated iteratively, until convergence or until a set amount of max iterations.

Hyper-parameter tuning of optimiser parameters is a necessary step before applying a NN to any problem. This is because, for larger data sets and for more iterations, one can save time by identifying the hyper-parameters that converge to the lowest possible NN cost in the least amount of time. Out of the three optimisers tested, it was seen that the ADAM optimiser led to faster convergence.

Although ADAM optimisation takes longer per iteration, the model with ADAM converges (finds the correct combination of NN weights and biases) 50 - 60 of iterations before the gradient descent models; Upon running this over and over again, this result is consistent. This finding is attributed to the fact that ADAM makes further use of the gradients of the model parameters, i.e. the first and second moments, which are second-order approximations of the gradient. One can argue that the fact that ADAM is an approximate second-order optimization algorithm, it is more efficient than first-order optimisation algorithms like gradient descent and gradient descent + momentum.