NITTE MEENAKSHI INSTITUTE OF TECHNOLOGY

(AN AUTONOMOUS INSTITUTION, AFFILIATED TO VISVESVARAYA TECHNOLOGICAL UNIVERSITY, BELGAUM, APPROVED BY AICTE & GOVT.OF KARNATAKA)

DEEP LEARNING COURSE PROJECT REPORT

titled

"NEURAL NETWORKS FROM THE SCRATCH"

Submitted in partial fulfilment of the requirement for the award of Degree of

Bachelor of Engineering

in

Computer Science and Engineering

Submitted by:

Harshith Narahari	**1NT15CS064**
Roshan Badrinath	**1NT15CS140**

Under the Guidance of

DR. SAROJADEVI H.
Professor, Dept. of CSE, NMIT

NITTE EDUCATION TRUST

Department of Computer Science and Engineering

2018-19

Nitte Meenakshi Institute of Technology

(AN AUTONOMOUS INSTITUTION AFFILIATED TO VISVESVARAYA TECHNOLOGICAL UNIVERSITY, BELGAUM)
PB No. 6429, Yelahanka, Bangalore 560-064, Karnataka
Telephone: 080- 22167800, 22167860
Fax: 080 – 22167805

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

This is to certify that the Deep Learning course project on "Neural Networks from the scratch" is an authentic work carried out by **Harshith Narahari (1NT15CS064)** and **Roshan Badrinath (1NT15CS140)** bona fide students of **Nitte Meenakshi Institute of Technology**, Bangalore in partial fulfilment for the award of the degree of ***Bachelor of Engineering*** in COMPUTER SCIENCE AND ENGINEERING of Visvesvaraya Technological University, Belagavi during the academic year ***2018-2019.***

Name and Signature of Guide	**Name and Signature of the HOD**
Dr. Saroja Devi. H	Dr. M. N. Thippeswamy

DECLARATION

We hereby declare that

(i) This Presentation does not contain text, graphics or tables copied and pasted from the Internet, unless specifically acknowledged, and the source being detailed in the report and in the References sections.

(ii) All corrections and suggestions indicated during the preparation have been incorporated in the report.

(iii) Content of the report has been checked for the plagiarism requirement

Name	USN	Signature
Harshith Narahari	1NT15CS064	
Roshan Badrinath	1NT15CS140	

Date:

ACKNOWLEDGEMENT

The satisfaction and euphoria that accompany the successful completion of any task would be incomplete without the mention of the people who made it possible, whose constant guidance and encouragement crowned our effort with success. We express our sincere gratitude to our Principal **Dr. H. C. Nagaraj**, Nitte Meenakshi Institute of Technology for providing facilities.

We thank our HoD, **Dr. Thippeswamy M.N.** for the support and encouragement to carry out this course project. We wish to express our gratitude to our project guide **Dr. Saroja Devi H.**, for teaching the course on Introduction to Deep Learning and to facilitate development of this project work on deep learning.

We also thank all our friends, teaching and non-teaching staff at NMIT, Bangalore, for all the direct and indirect help provided in the completion of the project work and this report.

Name	USN	Signature
Harshith Narahari	1NT15CS064	
Roshan Badrinath	1NT15CS140	

Date:

ABSTRACT

Artificial neural networks are computing systems that are used in machine learning and deep learning for regression and classification problems. Backpropagation is the algorithm that is used to train the neural network. The main aim of this algorithm is to get the best set of weights and biases that fit the dataset. Optimization is the process of updating weights efficiently. Gradient based optimization is the most popular optimization method.

In this project, a library, similar to *Keras API*, is developed in *Python* for implementing neural networks. The library provides functionality to create deep feed-forward neural networks with variable neurons in each layer. The library offers variety of activation functions and optimization functions.

The popular '*Iris flower dataset*' is used to test the performance of the neural network. Various optimization algorithms were used such as GD, SGD, NAM, AdaGrad, RMSProp and Adam. It was observed that NAM converges faster than SGD and Adam converges faster than AdaGrad and RMSProp.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ACRONYMS

Sl. No.	Acronym	Full Form
1.	AdaGrad	Adaptive Gradient
2.	Adam	Adaptive Moment Estimation
3.	AdaMax	Adam Maximum
4.	AI	Artificial Intelligence
5.	ANN	Artificial Neural Network
6.	CNN	Convolutional Neural Network
7.	DL	Deep Learning
8.	EWMA	Exponential Weighted Moving Average
9.	GD	Gradient Descent
10.	ML	Machine Learning
11.	MSE	Mean-Square Error
12.	Nadam	Nesterov and Adam
13.	NAM	Nesterov's Accelerated Momentum
14.	ReLU	Rectified Linear Unit
15.	RMSProp	Root Mean Square Propagation
16.	RNN	Recurrent Neural Network
17.	SGD	Stochastic Gradient Descent

1 INTRODUCTION

1.1 BACKGROUND

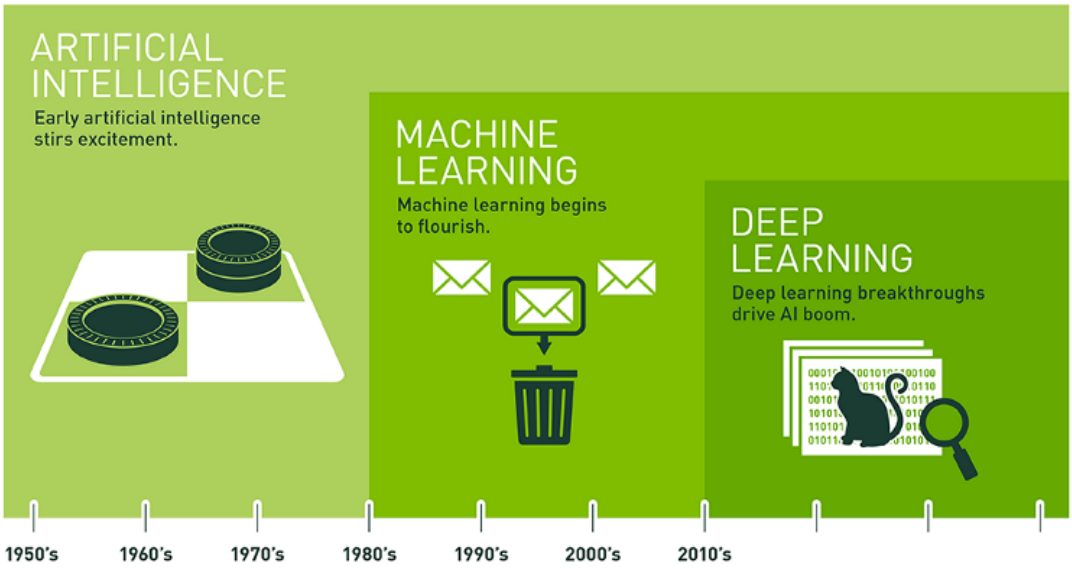

Fig 1.1: Relation between AI, ML and DL. [1]

Artificial intelligence is the intelligence displayed by machines in contrast to the natural intelligence displayed by humans and other animals. The field of AI is very broad and has been there for a long time. According to the *Fig 1.1*, the foundation of AI was laid in 1950. [1]

Machine Learning is a subset of AI, where computers systems are trained without any explicit set of instructions. An example of an ML task is an email spam filter. The ML algorithm is fed with large number of labeled email dataset. The main task of the algorithm is to find features to differentiate ‘spam’ and ‘not spam’. Then a set of unlabeled data is fed to the algorithm for prediction.

Deep Learning is a subset of ML, which is based on neural networks. Neural networks are conceptual model of the brain which has been around since 1950 but was largely ignored until 2010, as shown in *Fig 1.1*. Deep neural networks, i.e. more than 3 layers, are associated with deep learning. Deep learning also comprises of variants of neural networks such as CNN and RNN.

1.2 MOTIVATION

The learning process for a neural network is implemented by the backpropagation algorithm. This algorithm depends on the connections between successive layers, activation function for each layer and loss function of the neural network. The main task of this algorithm is to update the trainable parameters of the neural network. This process is also called as optimization. There are several optimization methods which have been implemented. A clear understanding of the performance of various optimization algorithm is necessary.

1.3 OBJECTIVES

The main objectives of the project would be:

- To create a library in Python to implement fully connected deep neural networks.
- To classify Iris flower dataset using an MLP.
- To compare the performance of various gradient based optimization methods such as GD, SGD, Nesterov's Accelerated Momentum, AdaGrad, RMSProp, and Adam.

1.4 ORGANIZATION OF THE REPORT

This project report contains 6 chapters. This introduction describes about the relation between AI, ML and DL and its evolution over the years. It also describes the motivation and objective of this project.

- Chapter 2 describes the literature survey which was undertaken in order to get the project started. It describes an ANN, training in ANN and different optimization algorithms.
- Chapter 3 describes the implementation details of the project. It provides information about the dataset and the implementation of the neural network library.
- Chapter 4 describes the results achieved using various optimization methods and some analysis based the results. It also compares different optimization methods which are implemented.

2 LITERATURE SURVEY

2.1 ARTIFICIAL NEUTRAL NETWORK

An Artificial Neural Network is an information processing paradigm and a computing system which is inspired by the way the biological nervous system, including the animal brain processes information. It consists of a large number highly interconnected processing units (also called neurons) which work together to achieve a common objective. Similar to people, ANNs learn by example. The advantageous features of ANNs are adaptive learning, self-organization, and real-time operational capability.

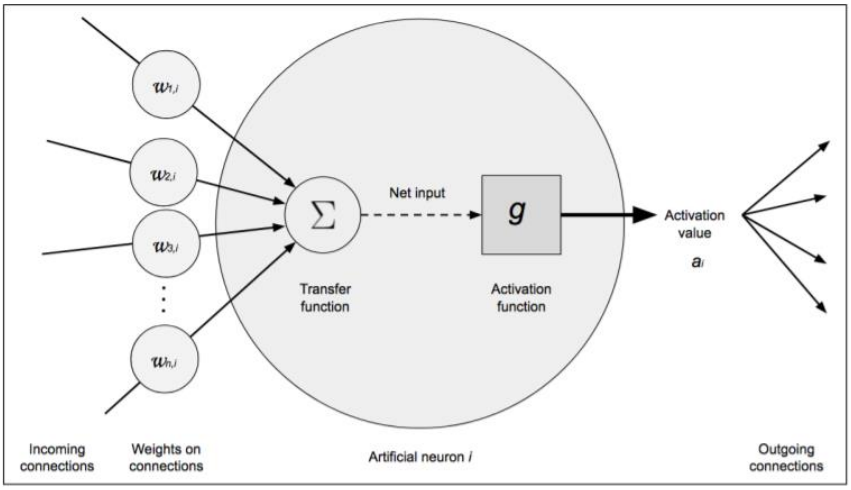

Fig 2.1: A perceptron model showing inputs, weights, net-sum and activation function [2].

Fig 2.1 shows a perceptron as a neuron where the inputs are associated with weights. It consists of 4 parts:

- Input values, which are inputs given to the neuron.
- Weights and bias, which represent the amount of importance each input value holds in determining the result
- Net Sum, which is the sum of all the input and weight product values.
- Activation Function, which is a node that has a normalising effect on the neuron output, thus helping to prevent a large value as output after moving through many layers of a neural network. It also helps the neuron to learn the non-linear and linear decision boundaries. *Sigmoid*, *ReLU* (Rectified Linear Unit) and *Tanh* are the most widely used activation functions.

ANNs can be extended to have many layers of different dimensions and structure. **Deep learning**, a part of ML, deals with such neural networks. CNNs and RNNs are examples of deep learning algorithms. *Fig 2.2* shows a simple feed-forward neural network consists of a single input layer, one or more hidden layers and an output layer, where each layer is made up of multiple neurons.

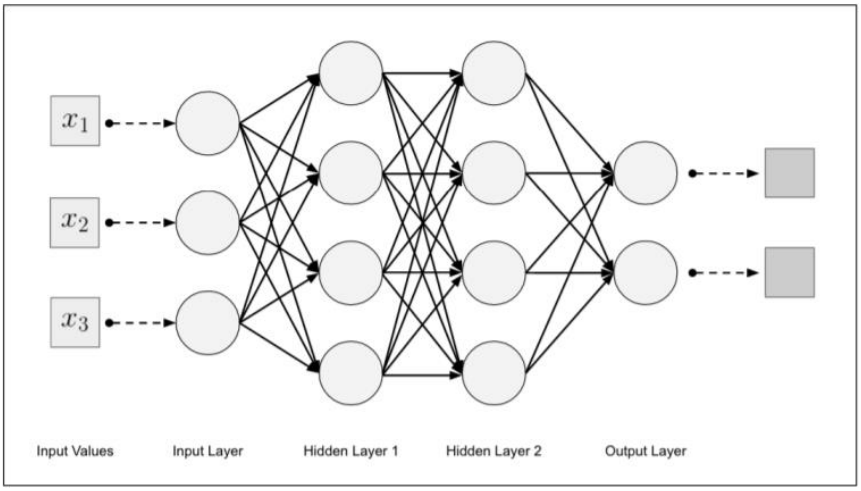

Fig 2.2: A feed-forward neural network with 2 hidden layer [2].

2.2 TRAINING A NEURAL NETWORK

Training in an artificial neural network is done using Backpropagation algorithm. The input to the algorithm is the dataset and the output is a trained neural network i.e. a neural network with updated weights. The output vector obtained after each forward movement is called the cost vector. A loss function (also called as cost function or error function) is used to find the error between the cost vector and theoretical values. *Fig 2.3* shows the backpropagation algorithm. The matrix which stores all the first order partial derivatives is called the Jacobian matrix (denoted by 'J'). The Jacobian depends on the input to the neuron and the partial derivative of the loss function (denoted by 'E') with respect to the net-sum (denoted by 'net'). The partial derivative is also called as gradient (denoted by 'δ', delta).

The gradient for the output layer '*j*' is as follows [3]:

$$\delta_j = \frac{\partial E}{\partial net_j} \frac{\partial net_j}{\delta w_{ij}}$$

The gradient for the hidden layer '*j*' is as follows [3]:

$$\delta_j = \sum_{k \, \epsilon \, all \, neurons \, in \, the \, next \, layer} \frac{\partial E}{\partial net_k} \frac{\partial net_k}{\partial net_j}$$

- x_{ji} = the ith input to unit j
- w_{ji} = the weight associated with the ith input to unit j
- $net_j = \sum_i w_{ji} x_{ji}$ (the weighted sum of inputs for unit j)
- o_j = the output computed by unit j
- t_j = the target output for unit j
- σ = the sigmoid function
- $outputs$ = the set of units in the final layer of the network
- $Downstream(j)$ = the set of units whose immediate inputs include the output of unit j

BACKPROPAGATION($training\_examples, \eta, n_{in}, n_{out}, n_{hidden}$)

Each training example is a pair of the form $\langle \vec{x}, \vec{t} \rangle$, where $\vec{x}$ is the vector of network input values, and $\vec{t}$ is the vector of target network output values.

η is the learning rate (e.g., .05). n_{in} is the number of network inputs, n_{hidden} the number of units in the hidden layer, and n_{out} the number of output units.

The input from unit i into unit j is denoted x_{ji}, and the weight from unit i to unit j is denoted w_{ji}.

- Create a feed-forward network with n_{in} inputs, n_{hidden} hidden units, and n_{out} output units.
- Initialize all network weights to small random numbers (e.g., between −.05 and .05).
- Until the termination condition is met, Do
  - For each $\langle \vec{x}, \vec{t} \rangle$ in $training\_examples$, Do

Propagate the input forward through the network:

1. Input the instance $\vec{x}$ to the network and compute the output o_u of every unit u in the network.

Propagate the errors backward through the network:

2. For each network output unit j, calculate its error term δ_j

$$\delta_j \leftarrow \frac{\partial E_d}{\partial o_j} \frac{\partial o_j}{\partial net_j} \quad \text{(T4.3)}$$

3. For each hidden unit j, calculate its error term δ_j

$$\delta_j \leftarrow \sum_{k \in Downstream(j)} \frac{\partial E_d}{\partial net_k} \frac{\partial net_k}{\partial net_j} \quad \text{(T4.4)}$$

4. Update each network weight w_{ji}

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

where

$$\Delta w_{ji} = \eta \, \delta_j \, x_{ji} \quad \text{(T4.5)}$$

Fig 2.3: Backpropagation algorithm [3]

2.3 OPTIMIZATION METHODS

Optimization methods are used to minimize the error of the loss function. This is used when the weights and biases are updated in the backpropagation algorithm. Gradient based optimization method is the most popular optimization method. It is a first order optimization methods.

2.3.1 GRADIENT DESCENT (GD)

Gradient descent is an optimization algorithm used to find minima of a function. It is used to update weights on a neural network. Gradient descent is the optimization method used in *Fig 2.3.* The weights are updated only after calculating gradient (δ) for all records of the dataset.

2.3.2 STOCHASTIC GRADIENT DESCENT (SGD)

The drawback of gradient descent is that it is time consuming as the weights are updated only after the gradient (δ) is calculated for all records in the dataset. If the size of the dataset is 'n', the time complexity of gradient descent is *O(n).*

Stochastic gradient descent is a variant of gradient descent algorithm. The weights are updated after calculating the gradient of each record in the dataset. The time complexity decreases to *O(1)* in SGD. If the dataset is very big, it is sometimes inefficient to calculate gradient (δ) for each record. The solution for this problem is to implement a *mini-batch SGD*. A batch comprises of certain number of records. The weights are updated after calculating gradient (δ) for each batch instead of each record.

The update rule for SGD is same as GD and is as follows:

$$w_{t+1} = w_t - \alpha J(w_t)$$

In the above equation '$J(w_t) = \frac{\partial L}{\partial w_t}$' is the Jacobian and 'α' is the learning late. As shown in *Fig 2.4*, over the years, many variant of SGD were developed to make the optimization more efficient.

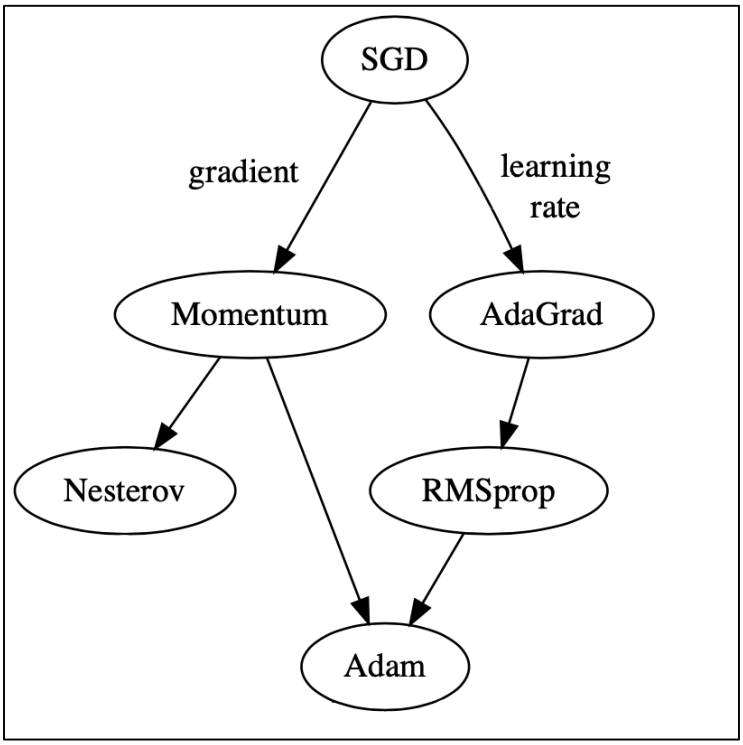

Fig 2.4: Evolution map of optimizers. [4]

2.3.3 MOMENTUM & NESTEROV'S ACCELERATED MOMENTUM (NAM)

The concept of *Exponential Weighted Moving Average* (EWMA) [5] is used to update the weights in gradient descent with momentum. EWMA is used to smoothen the minimization process and helps reach the minima efficiently. The update rule for gradient descent with momentum is as follows:

$$V_t = \beta V_{t-1} + (1-\beta) J(w_t)$$

$$w_{t+1} = w_t - \alpha V_t$$

The equation for 'V_t' is EWMA of the current and past gradient and is initialized to 0. It is observed that V_t and β are analogous to *Velocity* and *Momentum* in *Physics.*

Nesterov's Accelerated Momentum (NAM) [6] is slightly different from the vanilla momentum update method. Here, EWMA of *projected gradients* are calculated to further minimize oscillations while reaching the minima. The *Velocity* update rule is as follows:

$$V_t = \beta V_{t-1} + (1-\beta) J(w_t - \alpha V_{t-1})$$

The default value of β is 0.9.

2.3.4 ADAGRAD

The vanilla SGD has a uniform learning rate for the complete training process. It is better to have a larger learning rate during the beginning and smaller learning rate during the ending of the training process. AdaGrad divides the learning rate by the square root of cumulative sum of second order gradients. A small constant 'ε' (epsilon), also called as *fuzz factor,* is added to avoid division by zero. The constant 'ε' ranges from e^{-6} to e^{-8}. The update rule is as follows [4]:

$$S_t = S_{t-1} + J(w_t)^2$$

$$w_{t+1} = w_t - \frac{\alpha J(w_t)}{\sqrt{S_t} + \varepsilon}$$

The term 'S' is initialized to 0. The default value of 'α' is 0.01 and 'ε' is e^{-7}.

2.3.5 RMSPROP

In AdaGrad, it was observed that the learning rate always decreases. RMSProp is an extension of AdaGrad, where EWMA is used instead of cumulative sum of second order gradients. The 'S_t' update rule is as follows [4]:

$$S_t = \beta S_{t-1} + (1-\beta) J(w_t)^2$$

Here, the constant 'β' is the decay factor. It ranges from 0.9 to 0.999.

2.3.6 ADAM

An efficient optimization method can be derived from the combination of momentum based gradient descent and RMSProp. Adam is one such optimization method where EWMA of first order gradients (from momentum based GD) and second order gradients (from RMSProp) is considered. The terms 'V_t' and 'S_t' are called as first order moments and second order moments respectively [4].

EWMA fails to accurately predict gradient values initially due to initialization of 'V_t' and 'S_t' to 0. This can be prevented by adding a bias correction.

$$w_{t+1} = w_t - \frac{\alpha V_t'}{\sqrt{S_t'} + \varepsilon}$$

$$V_t' = \frac{V_t}{1-\beta_1^t} \qquad S_t' = \frac{S_t}{1-\beta_2^t}$$

$$V_t = \beta_1 V_{t-1} + (1-\beta_1) J(w_t) \qquad S_t = \beta_2 S_{t-1} + (1-\beta_2) J(w_t)^2$$

The terms 'V_t' and 'S_t' are initialized to 0. The constants β_1 and β_2 are momentum and decay factor respectively. They are initialized to 0.9 and 0.999 respectively. The terms V_t' and S_t' are bias correction terms.

3 IMPLEMENTATION DETAILS

3.1 DATASET DETAILS

The popular '*Iris flower dataset*' [7] is used in this project. The dataset consists of 150 records and 6 columns namely; *dataset order, sepal length, sepal width, petal length, petal width and species.*

The class variable is *species* with labels *Iris setosa, Iris virginica* and *Iris versicolor.* The dataset consists of 50 records for each species. The independent variables are *sepal length, sepal width, petal length, and petal width.* They are measured in centimeters. One class is linearly separable from the other 2, but other 2 are not. Hence a perceptron cannot be used for this classification problem.

3.2 NEURAL NETWORK IMPLEMENTATION

A library analogous to *Keras API* is implemented in *Python.* The library is named as '*neural_networks*'. The sub-modules are '*preprocessing*', '*activation*', '*loss*', '*training*' and '*optimization*'.

The '*preprocessing*' sub-module consists of a function to perform one-hot encoding. The sub-modules '*activation*' and '*loss*' consists of activation functions and its derivatives, and loss function and its derivatives respectively. The '*training*' sub-module provides functions to initialize vectors and calculate gradient (δ).

The '*optimization*' sub-module consist the implementation of various gradient based optimization methods such as GD, SGD, NAM, AdaGrad, RMSProp and Adam.

The '*neural_networks*' library consists of a class to create the neural network. It provides functionality to add multiple layers with variable neurons. The input layer is trivial and is not to be mentioned in the code. Instead the first hidden layer takes the dimension of the dataset as one its parameters, similar to *Keras API.* The library provides various activation functions such as *tanh, sigmoid, ReLU, and softmax.* The loss functions available are Mean-Square Error (MSE) and Cross-entropy.

As shows in *Fig 3.1,* Model class has 5 methods. After creating the object, multiple dense layers can be added sequentially. The input size has to be mentioned for the first layer. The last layer is the output layer and its size should be equal to the number of classes in the dataset.

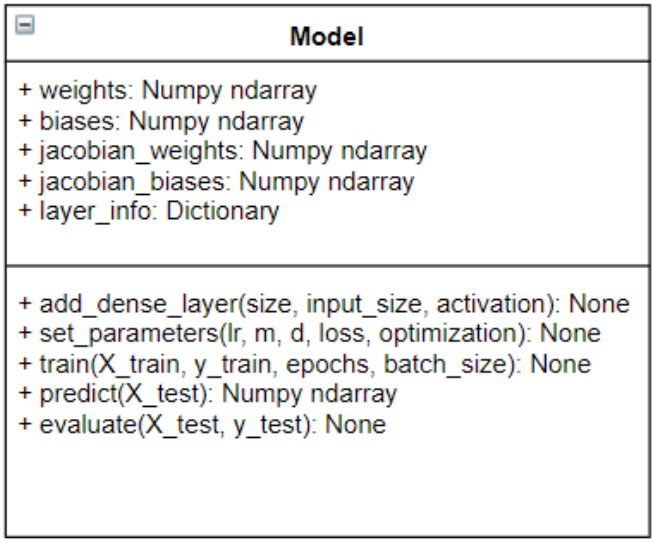

Fig 3.1: Class diagram for Model class

Dense neural network with 1 hidden layer

```
model = Model()

model.add_dense_layer(7, 4, activation='relu')
model.add_dense_layer(3, activation='softmax')
```

```
model.set_parameters(lr=0.01, loss='mse', optimization='gd')
```

```
model.train(X_train, y_train, 50, batch_size=12)
```

Fig 3.2: Code for creating a simple feed-forward neural network.

As shown in *Fig 3.2*, the parameters of '*set_parameters*' method has '*'lr*' as the learning rate, '*loss*' as the loss function and '*optimization*' as the optimization function. Additionally parameters are '*m*' as momentum or beta1 and '*d*' as decay factor beta2. The available loss functions are:

- MSE – '*mse*'
- Cross-entropy – '*cross_entropy*'

The available activation functions are:

- Tanh – '*tanh*'
- ReLU – '*relu*'
- Sigmoid – '*sigmoid*'
- Softmax – '*softmax*'

The available optimization functions are:

- Gradient descent – '*gd*'
- Stochastic gradient descent – '*sgd*'
- Nesterov's Aceelerated Momentum – '*nam*'
- AdaGrad – '*adagrad*'
- RMSProp – '*rmsprop*'
- Adam – '*adam*'

3.3 CODE OF OPTIMIZATION METHODS

3.3.1 GRADIENT DESCENT

```
# Gradient Descent (1 Batch)
def gd(self, X_train, y_train, epochs = 1):
    initialize_jacobian(self, X_train)
    X_train, y_train = create_batches(self, X_train, y_train, len(X_train))
    for epoch in range(epochs):
        for record, label in zip(X_train, y_train):
            evaluate_gradient(self, record, label)

        # Update weights and biases
        accumulated_jacobian_average(self, record)
        for i in range(self.num_layers-1, -1, -1):
            self.weights[i] += -self.lr * self.jacobian_weights[i]
            self.biases[i] += -self.lr * self.jacobian_biases[i]
        print_info(self, epoch)
```

3.3.2 STOCHASTIC GRADIENT DESCENT

```
# Stochastic Gradient Descent (mini batches)
def sgd(self, X_train, y_train, epochs, batch_size):
    x = X_train
    X_train, y_train, batches = create_batches(self, X_train, y_train,
batch_size)
    for epoch in range(epochs):
        for batch in range(batches):
            initialize_jacobian(self, x)
            for record, label in zip(X_train[batch], y_train[batch]):
                evaluate_gradient(self, record, label)

            # Update weights and biases
            accumulated_jacobian_average(self, record)
            for i in range(self.num_layers-1, -1, -1):
                self.weights[i] += -self.lr * self.jacobian_weights[i]
                self.biases[i] += -self.lr * self.jacobian_biases[i]
        print_info(self, epoch)
```

3.3.3 NESTEROV'S ACCELERATED MOMENTUM

```
# Nesterov's Accelerated Momentum
def nam(self, X_train, y_train, epochs, batch_size):
    x = X_train
    X_train, y_train, batches = create_batches(self, X_train, y_train,
batch_size)
    for epoch in range(epochs):
        for batch in range(batches):
            initialize_velocity(self, x)
            initialize_jacobian(self, x)
            for record, label in zip(X_train[batch], y_train[batch]):
                evaluate_gradient(self, record, label)

            # Update weights and biases with velocity
            # velocity_prev = velocity
            # velocity = m*velocity - lr*jacobian
            # weights += -m*velocity_prev + (1+m)*velocity
            accumulated_jacobian_average(self, record)
            for i in range(self.num_layers-1, -1, -1):
                # weights
                velocity_w = self.velocity_weights[i]
                self.velocity_weights[i] = (self.m *
self.velocity_weights[i]) - (self.lr * self.jacobian_weights[i])
                self.weights[i] += (-self.m * velocity_w) + ((1+self.m) *
self.velocity_weights[i])
                # biases
                velocity_b = self.velocity_biases[i]
                self.velocity_biases[i] = (self.m *
self.velocity_biases[i]) - (self.lr * self.jacobian_biases[i])
                self.biases[i] += (-self.m * velocity_b) + ((1+self.m) *
self.velocity_biases[i])
        print_info(self, epoch)
```

3.3.4 ADAGRAD

```
# Adagrad
def adagrad(self, X_train, y_train, epochs, batch_size):
    x = X_train
    eps = np.e**-7
    X_train, y_train, batches = create_batches(self, X_train, y_train,
batch_size)
    for epoch in range(epochs):
        for batch in range(batches):
            initialize_squared_gradient(self, x)
            initialize_jacobian(self, x)
            for record, label in zip(X_train[batch], y_train[batch]):
                evaluate_gradient(self, record, label)
```

```
            # Update weights and biases with sum of squared gradient
            # sum_squared_gradient += jacobian**2
            # weights += -lr*jacobian / (sqrt(sum_squared_gradient)+eps)
            accumulated_jacobian_average(self, record)
            for i in range(self.num_layers-1, -1, -1):
                # Second order gradient (jacobian**2)
                self.squared_gradient_weights[i] +=
self.jacobian_weights[i]**2
                self.squared_gradient_biases[i] +=
self.jacobian_biases[i]**2
                # First order gradient (jacobian)
                self.weights[i] += -self.lr * self.jacobian_weights[i] /
(np.sqrt(self.squared_gradient_weights[i]) + eps)
                self.biases[i] += -self.lr * self.jacobian_biases[i] /
(np.sqrt(self.squared_gradient_biases[i]) + eps)
        print_info(self, epoch)
```

3.3.5 RMSPROP

```
# RMSProp
def rmsprop(self, X_train, y_train, epochs, batch_size):
    x = X_train
    eps = np.e**-6
    X_train, y_train, batches = create_batches(self, X_train, y_train,
batch_size)
    for epoch in range(epochs):
        for batch in range(batches):
            initialize_squared_gradient(self, x)
            initialize_jacobian(self, x)
            for record, label in zip(X_train[batch], y_train[batch]):
                evaluate_gradient(self, record, label)

            # Update weights and biases with exponential moving average of
squared gradient
            # exp_moving_average_squared_gradient =
beta*exp_moving_average_squared_gradient + (1+beta)*jacobian**2
            # weights += -lr*jacobian /
(sqrt(exp_moving_average_squared_gradient)+eps)
            accumulated_jacobian_average(self, record)
            for i in range(self.num_layers-1, -1, -1):
                # Second order gradient (jacobian**2)
                self.squared_gradient_weights[i] = (self.d *
self.squared_gradient_weights[i]) + ((1-self.d) *
self.jacobian_weights[i]**2)
                self.squared_gradient_biases[i] = (self.d *
self.squared_gradient_biases[i]) + ((1-self.d) *
self.jacobian_biases[i]**2)
                # First oredr gradient (jacobian)
```

```
                self.weights[i] += -self.lr * self.jacobian_weights[i] /
(np.sqrt(self.squared_gradient_weights[i]) + eps)
                self.biases[i] += -self.lr * self.jacobian_biases[i] /
(np.sqrt(self.squared_gradient_biases[i]) + eps)
        print_info(self, epoch)
```

3.3.6 ADAM

```
# Adam
def adam(self, X_train, y_train, epochs, batch_size):
    x = X_train
    eps = np.e**-8
    t=0
    X_train, y_train, batches = create_batches(self, X_train, y_train,
batch_size)
    for epoch in range(epochs):
        t += 1
        for batch in range(batches):
            initialize_velocity(self, x)
            initialize_squared_gradient(self, x)
            initialize_jacobian(self, x)
            for record, label in zip(X_train[batch], y_train[batch]):
                evaluate_gradient(self, record, label)

            # Update weights and biases with velocity of gradient and
exponential moving average of squared gradient
            # velocity_gradient = m*velocity_gradient + (1+m)*jacobian
            # exp_moving_average_squared_gradient =
d*exp_moving_average_squared_gradient + (1+d)*jacobian**2
            # weights += -lr * velocity_gradient /
(sqrt(exp_moving_average_squared_gradient)+eps)
            accumulated_jacobian_average(self, record)
            for i in range(self.num_layers-1, -1, -1):
                # Second order gradient (jacobian**2)
                self.squared_gradient_weights[i] = (self.d *
self.squared_gradient_weights[i]) + ((1-self.d) *
self.jacobian_weights[i]**2)
                self.squared_gradient_biases[i] = (self.d *
self.squared_gradient_biases[i]) + ((1-self.d) *
self.jacobian_biases[i]**2)
                # Squared gradient corrected
                sw_corrected = self.squared_gradient_weights[i]/(1-
self.d**t)
                sb_corrected = self.squared_gradient_biases[i]/(1-
self.d**t)
                # First oredr gradient (jacobian)
                self.velocity_weights[i] = (self.m *
self.velocity_weights[i]) + ((1-self.m) * self.jacobian_weights[i])
```

```
                self.velocity_biases[i] = (self.m *
self.velocity_biases[i]) + ((1-self.m) * self.jacobian_biases[i])
                # Velocity correction
                vw_corrected = self.velocity_weights[i]/(1-self.m**t)
                vb_corrected = self.velocity_biases[i]/(1-self.m**t)
                # Update
                self.weights[i] += (-self.lr * vw_corrected /
(np.sqrt(sw_corrected) + eps))
                self.biases[i] += (-self.lr * vb_corrected /
(np.sqrt(sb_corrected) + eps))
        print_info(self, epoch)
```

4 RESULTS AND ANALYSIS

4.1 GRADIENT DESCENT AND STOCHASTIC GRADIENT DESCENT

```
model.set_parameters(lr=0.01, loss='mse', optimization='gd')
```

```
model.train(X_train, y_train, 50, batch_size=12)
```

```
Loss:  0.07590663924307713
Accuracy:  0.6916666666666667

Epoch  46
Loss:  0.0752850775026284
Accuracy:  0.6916666666666667

Epoch  47
Loss:  0.07467230756907993
Accuracy:  0.7

Epoch  48
Loss:  0.07406728429978113
Accuracy:  0.7

Epoch  49
Loss:  0.0734695752430517
Accuracy:  0.7
```

```
print(model.predict(X_test))
model.evaluate(X_test, y_test)
```

```
[2, 0, 0, 2, 2, 0, 2, 0, 2, 0, 2, 2, 2, 2, 2, 0, 1, 0, 2, 2, 0, 2, 0, 2, 0, 2, 2, 2]
Loss:  0.07586461044659526
Accuracy:  0.6666666666666666
```

```
model.set_parameters(lr=0.01, loss='mse', optimization='sgd')
```

```
model.train(X_train, y_train, 50, batch_size=12)
```

```
Loss:  0.07672903042176471
Accuracy:  0.65

Epoch  46
Loss:  0.07617849550933321
Accuracy:  0.65

Epoch  47
Loss:  0.07565222910976226
Accuracy:  0.65

Epoch  48
Loss:  0.07513934995941132
Accuracy:  0.65

Epoch  49
Loss:  0.07464959736353384
Accuracy:  0.65
```

```
print(model.predict(X_test))
model.evaluate(X_test, y_test)
```

```
[1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1]
Loss:  0.07001404953141808
Accuracy:  0.7333333333333333
```

Fig 4.1: (top) GD and (bottom) SGD

The effect SGD is generally observed when the size of the dataset is large. Hence, using SGD over GD for '*Iris flower dataset'* will not change the performance. As shown in *Fig 4.1*, the training accuracy of 0.7 and 0.65, and testing accuracy of 0.66 and 0.73 was achieved by using GD and SGD respectively.

4.2 NESTEROV'S ACCELERATED MOMENTUM

NAM is equivalent to SGD, if m=0

```
1 model.set_parameters(lr=0.01, m=0, loss='mse', optimization='nam')
```

```
1 model.train(X_train, y_train, 100, batch_size=12)
```

```
Epoch  60
Loss:  0.07314424735182594
Accuracy:  0.8833333333333333

Epoch  61
Loss:  0.07174127266240987
Accuracy:  0.9083333333333333

Epoch  62
Loss:  0.07041168201308896
Accuracy:  0.975

Epoch  63
Loss:  0.06915931667362106
Accuracy:  0.975

Epoch  64
Loss:  0.06798291198850746
Accuracy:  0.975
```

In NAM, 0.9<=m<=0.999

```
1 model.set_parameters(lr=0.01, m=0.9, loss='mse', optimization='nam')
```

```
1 model.train(X_train, y_train, 100, batch_size=12)
```

```
Epoch  16
Loss:  0.054208525919205545
Accuracy:  0.8666666666666667

Epoch  17
Loss:  0.05233401131960071
Accuracy:  0.8833333333333333

Epoch  18
Loss:  0.050560224975631576
Accuracy:  0.9

Epoch  19
Loss:  0.04880843697909216
Accuracy:  0.9083333333333333

Epoch  20
Loss:  0.04742202600123305
Accuracy:  0.9166666666666666
```

Fig 4.2: (top) SGD and (bottom) NAM

NAM is equivalent to SGD, if the value of '*m*' is 0. Both SGD and NAM reach high accuracy after certain epochs. As shown in *Fig 4.2*, SGD reached 90% accuracy at 64th epoch, whereas NAM reaches it in 18th epoch. Hence, NAM converges faster.

4.3 ADAGRAD, RMSPROP AND ADAM

```
1 model.set_parameters(lr=0.01, loss='mse', optimization='adagrad')
```

```
1 model.train(X_train, y_train, 50, batch_size=12)
```

```
Loss:  0.06649158139193094
Accuracy:  0.7

Epoch  46
Loss:  0.06636760245842026
Accuracy:  0.7

Epoch  47
Loss:  0.06624550842114957
Accuracy:  0.7

Epoch  48
Loss:  0.06612523553970552
Accuracy:  0.7

Epoch  49
Loss:  0.06600672312935418
Accuracy:  0.7
```

```
1 model.set_parameters(lr=0.01, d=0.9, loss='mse', optimization='rmsprop')
```

```
1 model.train(X_train, y_train, 100, batch_size=12)
```

```
Epoch  45
Loss:  0.04964559925461307
Accuracy:  0.7916666666666666

Epoch  46
Loss:  0.04966387784058469
Accuracy:  0.7916666666666666

Epoch  47
Loss:  0.049758812870807856
Accuracy:  0.7916666666666666

Epoch  48
Loss:  0.04978176652845448
Accuracy:  0.7916666666666666

Epoch  49
Loss:  0.049873225373174526
Accuracy:  0.7916666666666666
```

```
1 model.set_parameters(lr=0.01, m=0.9, d=0.999, loss='mse', optimization='adam')
```

```
1 model.train(X_train, y_train, 100, batch_size=12)
```

```
Epoch  45
Loss:  0.05003323890850898
Accuracy:  0.9333333333333333

Epoch  46
Loss:  0.04949854325537096
Accuracy:  0.9333333333333333

Epoch  47
Loss:  0.048932356856497786
Accuracy:  0.9333333333333333

Epoch  48
Loss:  0.048354640642276535
Accuracy:  0.9333333333333333

Epoch  49
Loss:  0.04778505792529773
Accuracy:  0.9333333333333333
```

Fig 4.3: (top) AdaGrad, (middle) RMSProp and (bottom) Adam

Adam is an extension of RMSProp and RMSProp is an extension of AdaGrad. As shown in *Fig 4.3*, after 50 iterations, accuracy of 0.7, 0.79 and 0.93 is achieved by using AdaGrad, RMSProp and Adam respectively. Hence, Adam converges faster than the others.

5 CONCLUSION

The objectives of this project was to develop a library, similar to *Keras API*, for implementing neural networks and to compare different optimization methods. A clear understanding of the training process of the neural network was necessary to develop the library. Different update rules were used by various optimization methods to make the learning process more efficient.

It was observed that for small dataset, the performance of GD and SGD were same. Adding momentum factor during the update rule for NAM helped the neural network converge faster when compared to SGD. Adam performed better than AdaGrad and RMSProp due to the consideration of both first order moments and second order moments.

The future implementation would include newer optimization methods such as Nadam, AMSGrad and AdaMax. The concept of *Regularization* can be introduced to increase the efficiency of the neural network.

6 REFERENCES

[1] "Deep learning weekly piece: the differences between AI, ML, and DL," Towards Data Science, 10 June 2017. [Online]. Available: https://towardsdatascience.com/deep-learning-weekly-piece-the-differences-between-ai-ml-and-dl-b6a203b70698.

[2] J. Patterson and A. Gibson, Deep Learning: A Practitioner's Approach, O'Reilly Media, 2017.

[3] T. M. Mitchell, Machine learning, McGraw Hill, 2017.

[4] "10 Gradient Descent Optimisation Algorithms," Towards Data Science, 22 November 2018. [Online]. Available: https://towardsdatascience.com/10-gradient-descent-optimisation-algorithms-86989510b5e9.

[5] A. Ng, "Exponentially weighted averages - Optimization algorithms," Coursera, [Online]. Available: https://www.coursera.org/lecture/deep-neural-network/exponentially-weighted-averages-duStO.

[6] CS231n, "Convolutional Neural Networks for Visual Recognition," Stanford University, [Online]. Available: http://cs231n.stanford.edu/.

[7] R. Fisher, "UCI Machine Learning Repository," [Online]. Available: http://archive.ics.uci.edu/ml/datasets/Iris.