
# Froppy Chess AI

## Version Alpha 1.0

Built By:
Abel Jimenez
Justin Lee
Khoi Trinh
Jason Duong
Aung Thu
Jeremiah Blackburn

Affiliated with: UCI

# Table of Contents

# Glossary

**Alpha-beta pruning**: search algorithm that tries to cut down the branches in the minimax algorithm search tree, making it faster.

**CLI:** command-line interface.

**CUDA:** An API as well as a platform that allows parallel computing using nvidia cards.

**FEN:** Forsyth–Edwards Notation, a standard used to record chess games after each move.

**LCG:** Linear congruential generator, an algorithm that recursively makes random numbers.

**NADAM:** An advanced gradient descent optimization algorithm that updates the gradient with with nesterov momentum and an adaptive learning rate.

**Neural Net:** A computer system modeled after the brain used to solve non-linear problems.

**Min-max** : an algorithm that finds a player's best move assuming the opponent plays optimally.

**OPENMP:** Open Multi-Processing, an API supporting shared memory parallel computing in CPU cores.

**RELU :** Rectified Linear unit used as activation format.

**RHEL**: Red Hat Enterprise Linux.

**SDL:** Simple DirectMedia Layer, a low-level API used for media interfaces.

**TANH:** A non-linear function that is used as an activation function.

**TD-Leaf:** An algorithm that applies temporal difference learning to a min-max leaf.

# 1 Software Architecture Overview

## 1.1 Main Data Types and Structures

**Boardstate.h**

```
/* This is the struct that hold all the board information */
typedef struct BSTATE;
```

**ChessGUI.h**

```
/* Global Variables for Screen settings  */
#define  WIDTH   640
#define  HEIGHT  540
#define  BPP         24

/* structs storing information from SDL interface */
SDL_Surface;
SDL_Event;
```

**minmax.h**

```
/* node structure to create n-ary trees for minmax */
typedef struct node NODE;

/* enum to define minimizers and maximizers */
typedef enum player{Min, Max} PLAYER;
```

**movelist.h**

```
/* The struct for move lists */
typedef struct MOVELIST MLIST;

/*The struct that contains each move */
typedef struct MOVEENTRY MENTRY
```

## 1.2. Major Software Components

Figure 1 below shows a diagram of the module hierarchy.

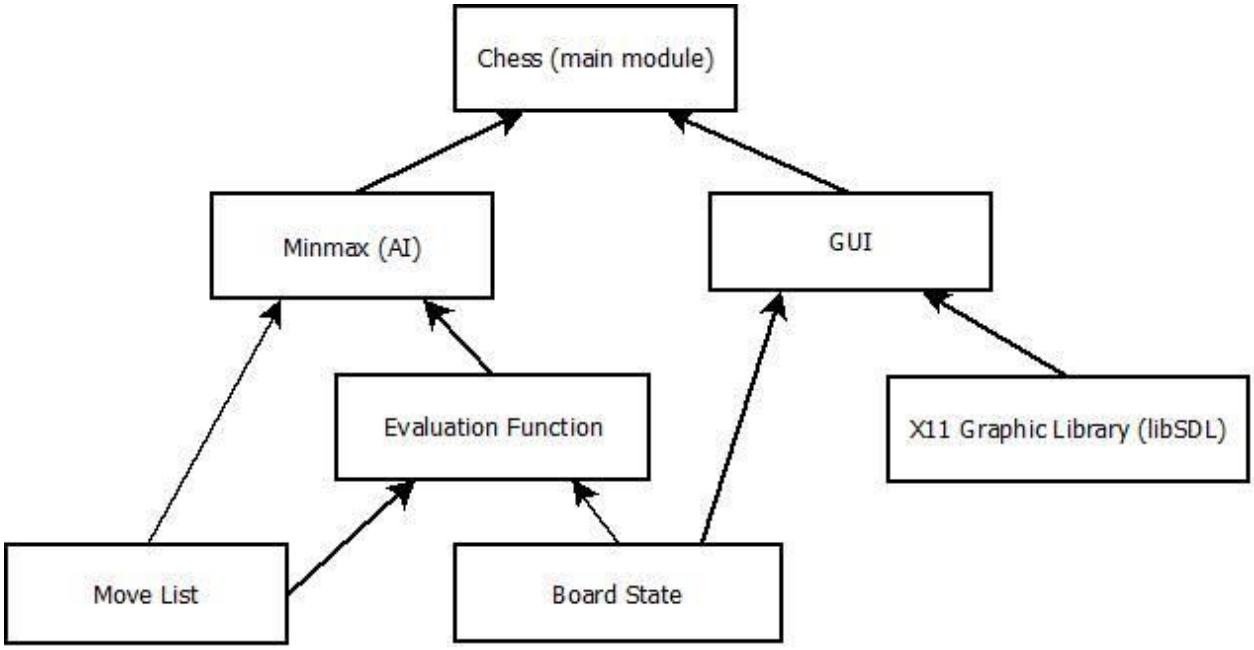


Figure 1. Diagram of Module Hierarchy

## 1.3. Module Interfaces

Below is an API of the major module functions:

**Chess Module**
Input: White or Black, Player Move,
Output: Board, AI Move
Description: Based on input, the module displays a board to the user. Then updates the board according to the moves made by player and AI.

**GUI Module**
Input: State of the board and the positions of the pieces
Output: Graphical representation of the board
Description: Creates an interface for the user to better interact with the chess program.

**Minmax (AI) Module**
Input: Current board.
Output: The best worst move for AI to make.

Description: creates a tree using the legal move list function. Then searches for the best worst path using minimax algorithm.

**Evaluation Function Module**
Input: Board
Output: Score for each board
Description: Evaluates the state of the input board and assigns a score to it.

**Move List Module**
Input: Board, player
Output: Linked list that contains the set of all possible legal moves
Description: Based on the state of input board and the player, the module creates a linked list containing all the legal moves for the player

**Board State Module**
Input: Move
Output: Board
Description: Creates a board structure, updates the board based on the input moves.

## 1.4. Overall Program Control Flow

Figure 2 below shows a diagram of the overall program flow.

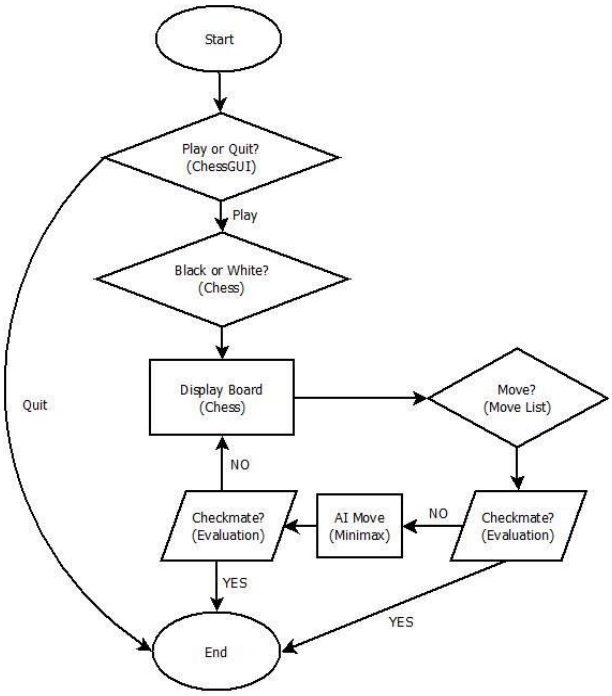


Figure 2. Diagram of overall program flow

## 1.5 Neural Network

Figure 3 below is a visual representation of the neural network; the numerical values represent the number of nodes in each layer.

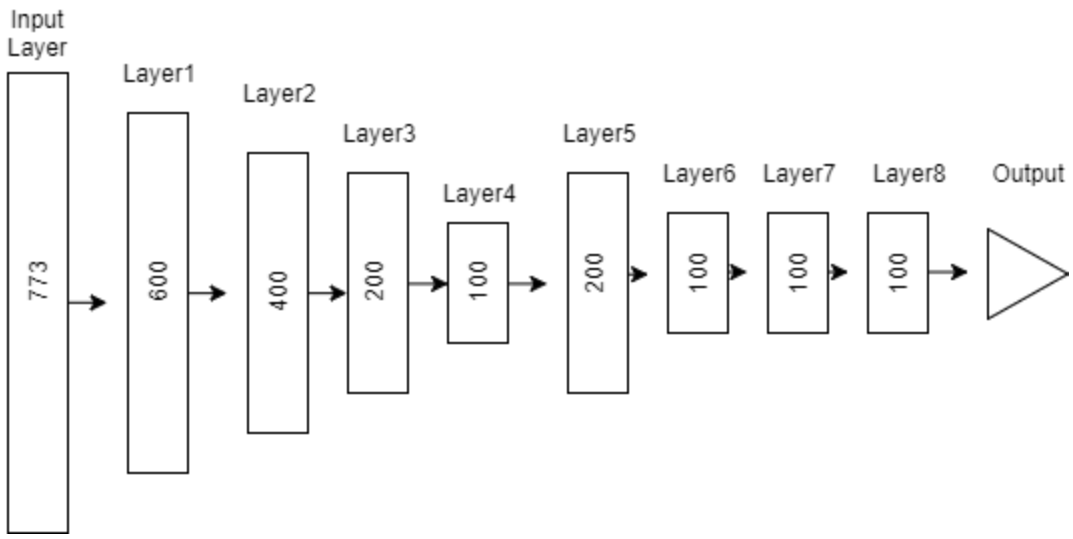


Figure 3. Neural network layer configuration.

Each layer is full connected. The first part of the neural network is an input vector that is 773 bits wide. The first 5 bits are the flags found in boardstate.h and the next 768 bits is the chessboard represented as a bitboard.

Layers 1-4 is a stacked auto-encoder that is trained greedily. The purpose of the auto-encoder is to lower the dimensionality of the input vector and at the same time raise the level of the features. Then layers 5-8 are used to find the strategy of the game chess through TD-Leaf.

The output layer is a tanh activation function so the output is restricted between -1 and 1. This works because chess is a zero sum game.

# 2 Installation

## 2.1 System Requirements & Compatibility

- Linux operating system (e.g. RHEL-6-x86_64)
- Gcc version 4.4.7 minimum
- Make utility

## 2.2 Setup & Configuration

- Ensure X11 Forwarding is enabled

## 2.3 Building, Compilation, & Installation

1) Download the source code file
2) Open a command prompt
3) Run "*make chess*"
4) Run the command "*./chess*"
5) Start playing!
6) Keep the command prompt open to see messages from the chess program.
7) To uninstall, run "*make uninstall*"

# 3 Documentation of Packages, Modules, & Interfaces

## 3.1 Detailed Description of Data Structures

This section details the data structures stored in the chess program's header files.

**boardstate.h**

```
/* This is the struct that hold all the board information */
typedef struct
{
        int boardarray[8][8];    /* stores the chess board in an 8x8 matrix */
        int WKCFlag;             /* flag specifying the white king */
        int WQCFlag;             /* flag specifying the white queen */
        int BKCFlag;             /* flag specifying the black king */
        int BQCFlag;             /* flag specifying the white queen */
        int sidetomove;          /* boolean value indicating who's move it is (white or black) */

} BSTATE;
```

**ChessGUI.h**

```
/* Global Variables for Screen settings  */
#define  WIDTH   640      /* int width of screen  */
#define  HEIGHT  540     /* int height of screen  */
#define  BPP         24       /* int bits per pixel      */

/* Contains the information on the screen's width and height (in pixel values), the format of the
pixels, and the pointer to the actual pixel data */
SDL_Surface;

/* Categorizes the user input by source. The main events the program will search for is mouse
activity such as SDL_MOUSEBUTTONDOWN or SDL_MOUSEMOTION  */
SDL_Event;
```

**minmax.h**

```
/* type definitions of the node and head in a n-ary tree */
typedef struct node NODE;
 typedef struct head HEAD;

/* structure to keep track of number of nodes in the tree */
struct head
{
   int length;              /* number of nodes in tree */
   NODE *root;          /* pointer to the root of tree */
}

/* node structure to create n-ary trees for minmax */
struct node
{
   float value;             /* the value of the board (after a move) from evaluation function */
   MENTRY *move;   /* the legal move used to initialize the node */
   BSTATE *board;     /* the state of the board after the move */
   NODE *parent;      /* pointer to parent node */
   NODE *child;        /* pointer to child node */
   NODE *next;         /* pointer to next sibling node */
   HEAD *head;        /* pointer to the head structure that keeps track of number of nodes */
};

/* enum to define minimizers and maximizers */
typedef enum player{Min, Max} PLAYER;
```

**movelist.h**

```
/* type definitions for move lists and move entries */
typedef struct MOVELIST MLIST;
typedef struct MOVEENTRY MENTRY;

/* The struct is for the list of move */
typedef struct MOVELIST
{
        int movenum;            /* number of moves made throughout a chess game */
        MENTRY *First;          /* first move made by a player in a chess game */
        MENTRY *Last;           /* last move made by a player in a chess game */
};

/*The struct that contains each move */
typedef struct MOVEENTRY
{
        int CLOC;               /* Current location in the array - indexed 1-64 */
        int NLOC;               /* New location in the array - indexed 1-64 */
        MLIST *List;            /* list of moves made by a player in the chess game log */
        MENTRY *Prev;           /* last move made by a player in the chess game log */
        MENTRY *Next;           /* next move made by a player in the chess game log */
};
```

## 3.2. Detailed Description of Functions & Parameters

This section provides the chess program's function prototypes along with brief explanations.

**ChessGUI.h**

void DisplayWindow(char *file, SDL_Surface *screen, int delay);
Inputs: filename, screen, amount of delay
Outputs: N/A
Description: Takes a .bmp file from a directory and copies it to a screen for some length of time.

void Add_Box(SDL_Surface *image, SDL_Surface *screen, int x, int y);
Inputs: 2 existing screens, x-y coordinate
Output: N/A
Description: Applies one screen onto another using the inputted coordinate as the reference point for the top left corner of the applied screen.

void Add_BoxFile(char *file, SDL_Surface *screen, int x, int y);
Inputs: filename, screen, x-y coordinates
Output: N/A
Description: In the case that the desired image is not already saved as an SDL_Surface, this function takes the .bmp file and applies it to the screen at the given coordinate. This is useful for when the the image is only used temporary or once and does not need to be saved as a surface.

void UpdateWindow(SDL_Surface *screen, int delay);
Inputs: screen, delay
Output: N/A
Description: Updates and displays the current screen and increases the amount of time it stays up. Used in while loops to continuously check for any changes in the screen.

void Exit(SDL_Surface *screen);
Inputs: screen
Output: N/A
Description: Frees the current working screen and follows SDL's exit protocol SDL_Quit().

**basic_eval.h** (basic evaluation function for testing purpose)

int basicEvaluation(BSTATE* currentboard);
Inputs: the state of the current board.
Outputs: the evaluation score of the current board.
Description: This function evaluates the current state of the board and give a rating of whether this would lead to a win or lose. This function is used for testing non-neural network features and will be replaced with neural network in the final release.

**fenToBoardState.h** (Converts a string formatted in fen and parses the information to the board state, and is reversible)

void fenToBoardState(char* fen, BSTATE *b);
Inputs: the FEN string as well as the BSTATE structure pointer that recieves the FEN information
Output: void.
Description: This function parses a FEN string and sends the information to a structure that holds the current state of the chess board.

void boardToFen(char *fen, BSTATE *b);
Inputs: The BSTATE pointer that represents the current board state, and the character string fen:
Output: void

Description: This function gets a BSTATE structure holding all the information on a given game, and parses it back into FEN, sending it into the character pointer fen;

**BoardState.h** (contains information on the board status)
The board is represented as a 8 by 8 array and each piece is encoded as an integer. The first digit represents the type of piece. It is represented as such: 1 is a pawn, 2 is a knight, 3 is a bishop, 4 is a rook, 5 is a queen, and 6 is the king. The next digit represents the type of piece 0 being white and 1 being black. i.e 11 is a black pawn and 1 is a white one.

void boardToVector(BSTATE *b, int *vector);
Inputs: BSTATE pointer as well as a int pointer named vector
Output: void
Description: This function will then get the BSTATE structure and parse it into a 1 dimensional long array to send it to the neural network. Converting the structure to the array makes it easier for the neural network to process information

**matrix.h** (provide matrix arithmetic functions)

Note: function with keyword __global__ can be executed both by CPU and CUDA GPU and is usually used for parallel computing.
__global__ void printMatrix(float *matrix, int width, int length);
Inputs: The matrix, and its width and length.
Outputs: The matrix shows up in the terminal.
Description: This function is used for debugging and will print out the input matrix. This function is global so that both CUDA code and CPU code can execute it.

__global__ void matrixMultiplication(float *h_a, float *h_b, float *h_result, int m, int n, int k);
Inputs: The two matrices that need to be multiplied, the matrix that the result will be saved as well as their dimensions.
Outputs: The results are saved into the supplied matrix.
Description: This function is used by CUDA GPU to multiply matrix together, there are usually multiple instances of this function running in parallel when the neural network is computing.

__global__ void transposeMatrix(float *inputmatrix, float *outputMatrix, const unsigned int row, const unsigned int column);
Inputs: The matrix to be transposed and the matrix that results will be saved to as well as the dimension of the input matrix.
Outputs: The result will be saved to the supplied output matrix.
Description: This function is used for CUDA GPU to transpose matrix and multiple instances of the function will run in parallel with each other to allow quick computing.

__global__ void combine3Matrix(float *a, float*b,float*c, float*combine, int rowa, int rowb, int rowc);
Inputs: This will concatenate three one dimension matrix together and needs the pointer that the new matrix will be pointed at, it also requires the dimension of the input matrix of the concatenation.
Outputs: The result will be saved to the supplied output matrix.
Description: This is used by CUDA GPU to join many matrices together. Many threads can execute multiple instances of this function in parallel.

**backprop.h** (provide back propagation functions)

__global__ void dererror_output_outer(float *nn_output, float *desired_output, *float der_result, int length);
Inputs: The array that contains the actual output the neural network, an array that contains the desired output of the neural network, an array to save the derivative of the error with respect to the output of the outermost layer as well as the length of the arrays.
Outputs: The result will be saved to the supplied output array.
Description: This function used by CUDA GPU calculates the derivative of the error with respect to the output of the outermost layer, then calculates the derivative of the error with respect to the value of the node. Many threads can execute multiple instances of this function in parallel.

__global__ void dererror_output(float *weight_next_layer, float *dererror_val, *float der_result, int length);
Inputs: The array that contains the weight of the next layer, an array that contains the derivative of the error with respect to the value of the next node, an array to save the derivative of the error with respect to the output of the layer as well as the length of the arrays.
Outputs: The result will be saved to the supplied output array.
Description: This function is used by CUDA GPU to calculate the derivative of the error with respect to the output of inner layers as a stepping stone for calculating the derivative of the error with respect to the value of the node. Many threads can execute multiple instances of this function in parallel.

__global__ void dererror_val(float *dererror_output, float *derror_val_res, int length);
Inputs: The derivative of the error with respect to the output of the node as well as an array to save the the derivative of the error with respect to the value of the node, the length of the array is also needed.
Outputs: The derivative of the error with respect to the value of the node is saved into the supplied array.

Description: This function is used by the CUDA GPU to compute the the derivative of the error with respect to the value, which is required to calculate the gradient descent. Many threads can execute multiple instances of this function in parallel.

__global__ void grad_des_comp(float *output_prev_layer, float *dererror_val, float learning_rate, float *grad_descrent, int length);
Inputs: the output of the previous layer, the derivative of the error with respect to value in each node as well as the learning rate of the neural network, an array that the gradient descent will be saved to is also needed, the length of the supplied result array is required.
Outputs: the calculated gradient descent will be saved in the supplied array.
Description: this function is executed by CUDA GPU to compute the gradient descent of weights in order to minimize the loss function. Many threads can execute multiple instances of this function in parallel.

**minmax.h** (finds the best worst move for AI to make)

NODE *createNode(float value, MENTRY *move);
Inputs: a legal move and the value of evaluated board after the move
Outputs: pointer to the created node
Description: The value and move is used to initialize the node (using malloc). Other pointers in the node struct are set to NULL.

NODE *addChild(NODE* parent, float value, BSTATE *board);
Inputs: parent node and the value and board to initialize the child node.
Outputs: pointer to child node
Description: Creates a child node pointer by calling createNode() with the inputs and sets the child pointer of parent node to the created node.

NODE *addSibling(NODE *child, float value, BSTATE *board);
Inputs: child(sibling) node and board to initialize the sibling node.
Outputs: pointer to sibling node
Description: Create a node by calling createNode() with inputs value and board. Then sets the next pointer of child to created sibling node.

void removeNode(NODE *node);
Inputs: node
Outputs: void
Description: deletes the node with its child and siblings by recursively calling free();

NODE *alphabeta(NODE* node, WEIGHTS *weights, float alpha, float beta, PLAYER minmax);

Inputs: node that contains current board, weights for evaluation function, alpha and beta values for pruning, PLAYER which defines if the node is a maximizer or minimizer.
Outputs: pointer to node that contains best worst move.
Description: Use the minimax with alpha-beta pruning algorithm to find the node with best worst float value in the first layer of the tree. Then it returns the pointer to that node.

/* MENTRY *minmax (BSTATE *currentBoard, WEIGHTS *weights) */
Inputs: current board and weight for evaluation function.
Outputs: move entry that contains the best worst move.
Description: creates the tree by calling the function that finds all legal moves in a loop. This loop is stopped by time. After the tree is created, alphabeta() is called. The pointer to MENTRY inside the node returned by alphabeta() is returned from this function.

## 3.3. Detailed Description of Input and Output Functions

### *Syntax/Format of a move input by user*

When it is the user's turn, they are prompted to input a move in the form of **[current coordinate][next coordinate]**. Whichever piece is in the [current coordinate] position, it is moved to the [next coordinate] position as long as it is a legal move, evaluated by the function that lists all legal moves at that current board state. An example of a user prompt and input for moving a black pawn forward one square from board position e7 to e6 would be:

"It is now your turn! Please input move: *e7e6*"

### *Syntax/Format of a move recorded in the log file*

After a user inputs a legal move, the move is appended to the move log. Using the same example as above, the log entry would look like:

"Black side moved *e7* to *e6*"

# 4 Development Plan/Timeline

**Week 1**: Get together and figure out the game plan.
**Week 2**: Foundation code for the structures and main files should be completed.
**Week 3**: The whole program should be ready to run for the training for the first time.
**Week 4**: Extra features should be implemented.
**Week 5**: Program should be fully trained and ready for full release.

## 4.1. Task Partitioning

**Stand Alone Chess Program:**
This will be the Standard Chess Program with a GUI and a AI to play against.

**Training-Data Parser:**
This program will take in games in pgn form and randomly select a position in the game then add a random move to it and save it to a text file as a fen for input into the training program.

**Training Program:**
This program will take the training data to train the neural network.

**GUI:**
Interface for ease of use on human-side operations.
**Neural Net architecture:**
The neural net is made up of two parts: stacked autoencoder and a feed-forward network. The input is a vectorized bitboard with the 4 castle flags and 1 side to move flag. A stacked autoencoder is then trained to compress the data. Then after it is trained, it is connected to a feed-forward network that is then refined with back propagation. Additional code is written for matrix operations used for arithmetic operation during neural network training. To speed up the training, most of the arithmetic is done with CUDA GPU with computations typically running asynchronously parallel.

## 4.2. Individual Responsibilities

Abel Jimenez: Create the structure for the MoveList and for the boardstate. Code the piece movement into the game, design the neural net, wrote dpacker, and nadam functions.

Justin Lee: All around jack of all trades, will work on openmp/cuda/sdl implmentations as well as converting Fen to usable data for the neural net.

Khoi Trinh: Backward Propagation, Basic Evaluation Function and Matrix Arithmetic using CUDA.

Jason Duong: ChessGUI (using SDL 1.2) and main().

Aung Thu: Minimax with alpha-beta pruning search algorithm used to find the best-worst move for AI.

Jeremiah Blackburn: neural network initialization.

# Copyright



# References

1.“CUDA C Programming Guide.” CUDA Toolkit Documentation, NVIDIA Corporation, 19 Dec. 2017, <docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>.

2. Liu, Jinfeng, and Lei Guo. “Implementation of Neural Network Backpropagation in CUDA.” SpringerLink, Springer, Berlin, Heidelberg, 1 Jan. 1970, <link.springer.com/chapter/10.1007/978-3-642-31656-2_140>.

3. Matthew Lai. “Giraffe: Using Deep Reinforcement learning to play Chess” Imperial College London September 2015 <https://arxiv.org/pdf/1509.01549v2.pdf>.

4. Jonathan Baxter, Andrew Tridgell, and Lex Weaver. “KnightCap: A chess program that learns by combining TD(λ) with game-tree search” Australian National University, 10 January 1999, <https://arxiv.org/pdf/cs/9901002.pdf>.

5. Diederik P. Kingma, and Jimmy Lei Ba “ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION”, University of Amsterdam ICLR 2015 ,<https://arxiv.org/pdf/1412.6980.pdf>.

6. Xavier Glorot, and Yoshua Bengio “Understanding the difficulty of training deep feedforward neural networks” DIRO, Universite de Montreal, Montreal, Quebec, Canada <http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf>.

# Index