cgrad

Generated by Doxygen 1.10.0

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 3

Class Documentation

3.1 ann_struct Struct Reference

A feedforward neural network.

`#include <ann.h>`

Public Attributes

- int **n_layers**
- LAYER ** **layers**

3.1.1 Detailed Description

A feedforward neural network.

The documentation for this struct was generated from the following file:

- nn/ann.h

3.2 layer_struct Struct Reference

A layer of neurons in a feedforward neural network.

`#include <layer.h>`

Public Attributes

- int **size**
- NEURON ** **neurons**
- OPERATION **activation**

3.2.1 Detailed Description

A layer of neurons in a feedforward neural network.

The documentation for this struct was generated from the following file:

- nn/layer.h

3.3 neuron_struct Struct Reference

A neuron in a neural network.

`#include <neuron.h>`

Public Attributes

- int **num_inputs**
- PARAM * **params**
- VALUE ** **weights**

3.3.1 Detailed Description

A neuron in a neural network.

The documentation for this struct was generated from the following file:

- nn/neuron.h

3.4 node_struct Struct Reference

A node in a linked list.

`#include <grad.h>`

Public Attributes

- VALUE * **value**
- struct node_struct * **next**

3.4.1 Detailed Description

A node in a linked list.

The documentation for this struct was generated from the following file:

- utils/grad.h

3.5 param_struct Struct Reference

A parameter of a neuron in a neural network.

`#include <grad.h>`

Public Attributes

- double **val**
- double **grad**
- double **momentum**

3.5.1 Detailed Description

A parameter of a neuron in a neural network.

The documentation for this struct was generated from the following file:

- utils/grad.h

3.6 value_struct Struct Reference

A singe unit/value in the computational graph during backpropagation.

`#include <grad.h>`

Public Attributes

- double **val**
- double **grad**
- void(∗ **backward**)(struct value_struct ∗)
- OPERATION **op**
- struct value_struct ∗ **left**
- struct value_struct ∗ **right**
- bool **visited**
- PARAM ∗ **param**

3.6.1 Detailed Description

A singe unit/value in the computational graph during backpropagation.

The documentation for this struct was generated from the following file:

- utils/grad.h

Chapter 4

File Documentation

4.1 load/data.c File Reference

Functions to load and free data from the MNIST dataset.

```
#include "data.h"
```

Functions

- void print_image (double *image, int label)
  Prints the image to the console.
- double * read_image (FILE *file, int *label)
  Reads a single image from the file.
- void read_csv (char *path, double ***images_addr, int **labels_addr, int size)
  Reads the CSV file and stores the images and labels.
- void free_images (double **images, int *labels, int size)
  Frees the memory allocated to the images.

4.1.1 Detailed Description

Functions to load and free data from the MNIST dataset.

This contains the implementations for the functions used to read and free data from the MNIST dataset in CSV format. It is used by the neural network in `mnist.c`.

Author

Vamsi Deeduvanu (vamsi10010)

4.1.2 Function Documentation

4.1.2.1 free_images()

```
void free_images (
            double ** images,
            int * labels,
            int size )
```

Frees the memory allocated to the images.

Parameters

images	Pointer to the 2D array with flattened images
labels	Pointer to the array with labels of images
size	Number of images

4.1.2.2 print_image()

```
void print_image (
            double * image,
            int label )
```

Prints the image to the console.

Parameters

image	Array of doubles corresponding to pixels of flattened image
size	Number of pixels in the image

Returns

void

4.1.2.3 read_csv()

```
void read_csv (
            char * path,
            double *** images_addr,
            int ** labels_addr,
            int size )
```

Reads the CSV file and stores the images and labels.

Parameters

filename	Name of the CSV file with MNIST data
images	Pointer to the 2D array to store flattened images
labels	Pointer to the array to store labels of images
size	Number of images in the CSV file

4.1.2.4 read_image()

```
double * read_image (
            FILE * file,
            int * label )
```

Reads a single image from the file.

Parameters

fp	File pointer to the CSV file with MNIST data
size	Pointer to the variable to store the label of the image

Returns

Array of doubles corresponding to pixels of flattened image

4.2 load/data.h File Reference

Function prototypes for data.c.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "../utils/grad.h"
```

Macros

- #define **PIXELS** 784
- #define **LABELS** 10
- #define **TRAIN_SIZE** 60000
- #define **TEST_SIZE** 10000
- #define **TRIAL_SIZE** 10
- #define **TRAIN_IMAGES** "./data/mnist_train.csv"
- #define **TEST_IMAGES** "./data/mnist_test.csv"
- #define **TRIAL_IMAGES** "./data/mnist_trial.csv"

Functions

- void print_image (double *image, int label)

  Prints the image to the console.
- double * read_image (FILE *file, int *label)

  Reads a single image from the file.
- void read_csv (char *path, double ***images_addr, int **labels_addr, int size)

  Reads the CSV file and stores the images and labels.
- void free_images (double **, int *, int)

  Frees the memory allocated to the images.

4.2.1 Detailed Description

Function prototypes for data.c.

This contains the function prototypes for loading data from the MNIST dataset in CSV format.

Author

Vamsi Deeduvanu (vamsi10010)

4.2.2 Function Documentation

4.2.2.1 free_images()

```
void free_images (
            double ** images,
            int * labels,
            int size )
```

Frees the memory allocated to the images.

Parameters

images	Pointer to the 2D array with flattened images
labels	Pointer to the array with labels of images
size	Number of images

4.2.2.2 print_image()

```
void print_image (
            double * image,
            int label )
```

Prints the image to the console.

Parameters

image	Array of doubles corresponding to pixels of flattened image
size	Number of pixels in the image

Returns

void

4.2.2.3 read_csv()

```
void read_csv (
            char * path,
            double *** images_addr,
            int ** labels_addr,
            int size )
```

Reads the CSV file and stores the images and labels.

Parameters

filename	Name of the CSV file with MNIST data
images	Pointer to the 2D array to store flattened images
labels	Pointer to the array to store labels of images
size	Number of images in the CSV file

4.2.2.4 read_image()

```
double * read_image (
            FILE * file,
            int * label )
```

Reads a single image from the file.

Parameters

fp	File pointer to the CSV file with MNIST data
size	Pointer to the variable to store the label of the image

Returns

Array of doubles corresponding to pixels of flattened image

4.3 data.h

Go to the documentation of this file.

```
00001
00010 #ifndef __DATA_H__
00011 #define __DATA_H__
00012
00013 #include <stdio.h>
00014 #include <stdlib.h>
00015 #include <string.h>
00016
00017 #include "../utils/grad.h"
00018
00019 #define PIXELS 784
00020 #define LABELS 10
00021
00022 #define TRAIN_SIZE 60000
00023 #define TEST_SIZE 10000
00024 #define TRIAL_SIZE 10
00025
00026 #define TRAIN_IMAGES "./data/mnist_train.csv"
00027 #define TEST_IMAGES "./data/mnist_test.csv"
00028 #define TRIAL_IMAGES "./data/mnist_trial.csv"
00029
00036 void print_image(double *image, int label);
00037
00044 double *read_image(FILE *file, int *label);
00045
00053 void read_csv(char *path, double ***images_addr, int **labels_addr, int size);
00054
00061 void free_images(double **, int *, int);
00062
00063 #endif // __DATA_H__
```

4.4 main/mnist.c File Reference

Trains and tests a feedforward neural network on the MNIST dataset.

```
#include "mnist.h"
```

Functions

- int * perm (int n)

  Generates a random permutation of the integers from 0 to n - 1.
- void train (ANN *nn)

  Trains the neural network on the MNIST dataset.
- void test (ANN *nn)

  Tests the neural network on the MNIST dataset.
- int **main** ()

4.4.1 Detailed Description

Trains and tests a feedforward neural network on the MNIST dataset.

This contains the implementation for the functions used to train and test a feedforward neural network on the MNIST dataset.

Author

Vamsi Deeduvanu (vamsi10010)

4.4.2 Function Documentation

4.4.2.1 perm()

```
int * perm (
            int n )
```

Generates a random permutation of the integers from 0 to n - 1.

Parameters

n	The number of integers to permute.

Returns

A pointer to the array of integers.

4.4.2.2 test()

```
void test (
            ANN * nn )
```

Tests the neural network on the MNIST dataset.

Parameters

nn	The neural network.

4.4.2.3 train()

```
void train (
            ANN * nn )
```

Trains the neural network on the MNIST dataset.

Parameters

nn	The neural network.

4.5 main/mnist.h File Reference

Trains and tests a feedforward neural network on the MNIST dataset.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include "../nn/ann.h"
#include "../load/data.h"
```

Macros

- #define **NUM_LAYERS** 3
- #define **LEARNING_RATE** 0.1
- #define **EPOCHS** 3
- #define **MOM_COEFF** 0.9
- #define **REG_COEFF** 0.1
- #define **BATCH_SIZE** 32
- #define **OUTPUT_SIZE** 10

Functions

- int * perm (int n)

  Generates a random permutation of the integers from 0 to n - 1.
- void train (ANN *nn)

  Trains the neural network on the MNIST dataset.
- void test (ANN *)

  Tests the neural network on the MNIST dataset.

4.5.1 Detailed Description

Trains and tests a feedforward neural network on the MNIST dataset.

This contains the prototypes for the functions used to train and test a feedforward neural network on the MNIST dataset.

Author

Vamsi Deeduvanu (vamsi10010)

4.5.2 Function Documentation

4.5.2.1 perm()

```
int * perm (
            int n )
```

Generates a random permutation of the integers from 0 to n - 1.

Parameters

n	The number of integers to permute.

Returns

A pointer to the array of integers.

4.5.2.2 test()

```
void test (
            ANN * nn )
```

Tests the neural network on the MNIST dataset.

Parameters

nn	The neural network.

4.5.2.3 train()

```
void train (
            ANN * nn )
```

Trains the neural network on the MNIST dataset.

Parameters

nn	The neural network.

4.6 mnist.h

Go to the documentation of this file.

```
00001
00010 #ifndef __MNIST_H__
00011 #define __MNIST_H__
00012
00013 #include <stdio.h>
00014 #include <stdlib.h>
```

```
00015 #include <string.h>
00016 #include <stdbool.h>
00017
00018 #include "../nn/ann.h"
00019 #include "../load/data.h"
00020
00021 #define NUM_LAYERS 3
00022 #define LEARNING_RATE 0.1
00023 #define EPOCHS 3
00024 #define MOM_COEFF 0.9
00025 #define REG_COEFF 0.1
00026 #define BATCH_SIZE 32
00027 #define OUTPUT_SIZE 10
00028
00034 int *perm(int n);
00035
00040 void train(ANN *nn);
00041
00046 void test(ANN *);
00047
00048 #endif // __MNIST_H__
```

4.7 nn/ann.c File Reference

Feedforwar neural network implementation.

```
#include "ann.h"
```

Functions

- ANN * ann (int num_layers, int *layer_sizes, OPERATION *activations, int num_inputs)
  Creates a feedforward neural network.
- VALUE ** ann_forward (ANN *a, VALUE **x)
  Performs a forward pass on the network.
- double * ann_nograd_forward (ANN *a, double *x)
  Performs a forward pass on the network without creating a graph.
- VALUE * regularization (ANN *a, REG reg, double c)
  Calculates the regularization term.
- void ann_descend (ANN *a, double lr, bool momentum)
  Performs a gradient descent step on the network.
- void free_ann (ANN *a)
  Frees the memory allocated to the network.
- void zero_grad (ANN *a)
  Sets the gradients of the network to zero.
- VALUE * loss_fn (VALUE **yhat, double y, LOSS loss, int size)
  Calculates the loss of the network (classification only).
- double loss_fn_nograd (double *yhat, double y, LOSS loss, int size)
  Calculates the loss of the network (classification only) without values.
- int predict (ANN *n, double *x, int classes)
  Predicts the class of an input.

4.7.1 Detailed Description

Feedforwar neural network implementation.

This contains the implementation for the functions used to create a feedforward neural network.

Author

Vamsi Deeduvanu (vamsi10010)

4.7.2 Function Documentation

4.7.2.1 ann()

```
ANN * ann (
            int num_layers,
            int * layer_sizes,
            OPERATION * activations,
            int num_inputs )
```

Creates a feedforward neural network.

Parameters

num_layers	The number of layers in the network.
layer_sizes	The sizes of the layers in the network.
activations	The activation functions for the layers in the network.
num_inputs	The number of inputs to the network.

Returns

A pointer to the network.

4.7.2.2 ann_descend()

```
void ann_descend (
            ANN * a,
            double lr,
            bool momentum )
```

Performs a gradient descent step on the network.

Parameters

a	The network.
lr	The learning rate.
momentum	Whether to use momentum.

4.7.2.3 ann_forward()

```
VALUE ** ann_forward (
            ANN * a,
            VALUE ** x )
```

Performs a forward pass on the network.

Parameters

a	The network.
x	The input to the network.

Returns

The output of the network.

4.7.2.4 ann_nograd_forward()

```
double * ann_nograd_forward (
            ANN * a,
            double * x )
```

Performs a forward pass on the network without creating a graph.

Parameters

a	The network.
x	The input to the network.

Returns

The output of the network.

4.7.2.5 free_ann()

```
void free_ann (
            ANN * a )
```

Frees the memory allocated to the network.

Parameters

a	The network.

4.7.2.6 loss_fn()

```
VALUE * loss_fn (
            VALUE ** yhat,
            double y,
            LOSS loss,
            int size )
```

Calculates the loss of the network (classification only).

Parameters

yhat	The output of the network.
y	The target class.
loss	The loss function.
size	The size of the output.

Returns

The loss.

4.7.2.7 loss_fn_nograd()

```
double loss_fn_nograd (
            double * yhat,
            double y,
            LOSS loss,
            int size )
```

Calculates the loss of the network (classification only) without values.

Parameters

yhat	The output of the network.
y	The target class.
loss	The loss function.
size	The size of the output.

Returns

The loss.

4.7.2.8 predict()

```
int predict (
            ANN * n,
            double * x,
            int classes )
```

Predicts the class of an input.

Parameters

n	The network.
x	The input.
classes	The number of classes.

Returns

The predicted class.

4.7.2.9 regularization()

```
VALUE * regularization (
            ANN * a,
```

```
            REG reg,
            double c )
```

Calculates the regularization term.

Parameters

a	The network.
reg	The regularization type.
c	The regularization coefficient.

4.7.2.10 zero_grad()

```
void zero_grad (
            ANN * a )
```

Sets the gradients of the network to zero.

Parameters

a	The network.

4.8 nn/ann.h File Reference

Header file for a feedforward neural network.

```
#include "../utils/grad.h"
#include "layer.h"
```

Classes

- struct ann_struct

  A feedforward neural network.

Typedefs

- typedef enum loss_enum **LOSS**

  Loss functions.
- typedef struct ann_struct **ANN**

  A feedforward neural network.

Enumerations

- enum loss_enum { **MSE** , **CROSS_ENTROPY** }

  Loss functions.

Functions

- ANN * ann (int num_layers, int *layer_sizes, OPERATION *activations, int num_inputs)
  Creates a feedforward neural network.
- VALUE ** ann_forward (ANN *a, VALUE **x)
  Performs a forward pass on the network.
- VALUE * regularization (ANN *a, REG reg, double c)
  Calculates the regularization term.
- void ann_descend (ANN *a, double lr, bool momentum)
  Performs a gradient descent step on the network.
- void free_ann (ANN *a)
  Frees the memory allocated to the network.
- void zero_grad (ANN *a)
  Sets the gradients of the network to zero.
- VALUE * loss_fn (VALUE **yhat, double y, LOSS loss, int size)
  Calculates the loss of the network (classification only).
- double * ann_nograd_forward (ANN *a, double *x)
  Performs a forward pass on the network without creating a graph.
- double loss_fn_nograd (double *yhat, double y, LOSS loss, int size)
  Calculates the loss of the network (classification only) without values.
- int predict (ANN *n, double *x, int classes)
  Predicts the class of an input.

4.8.1 Detailed Description

Header file for a feedforward neural network.

This contains the prototypes for the functions used to create a feedforward neural network.

Author

Vamsi Deeduvanu (vamsi10010)

4.8.2 Function Documentation

4.8.2.1 ann()

```
ANN * ann (
            int num_layers,
            int * layer_sizes,
            OPERATION * activations,
            int num_inputs )
```

Creates a feedforward neural network.

Parameters

num_layers	The number of layers in the network.
layer_sizes	The sizes of the layers in the network.
activations	The activation functions for the layers in the network.
num_inputs	The number of inputs to the network.

Returns

A pointer to the network.

4.8.2.2 ann_descend()

```
void ann_descend (
            ANN * a,
            double lr,
            bool momentum )
```

Performs a gradient descent step on the network.

Parameters

a	The network.
lr	The learning rate.
momentum	Whether to use momentum.

4.8.2.3 ann_forward()

```
VALUE ** ann_forward (
            ANN * a,
            VALUE ** x )
```

Performs a forward pass on the network.

Parameters

a	The network.
x	The input to the network.

Returns

The output of the network.

4.8.2.4 ann_nograd_forward()

```
double * ann_nograd_forward (
            ANN * a,
            double * x )
```

Performs a forward pass on the network without creating a graph.

Parameters

a	The network.
x	The input to the network.

Returns

The output of the network.

4.8.2.5 free_ann()

```
void free_ann (
            ANN * a )
```

Frees the memory allocated to the network.

Parameters

a	The network.

4.8.2.6 loss_fn()

```
VALUE * loss_fn (
            VALUE ** yhat,
            double y,
            LOSS loss,
            int size )
```

Calculates the loss of the network (classification only).

Parameters

yhat	The output of the network.
y	The target class.
loss	The loss function.
size	The size of the output.

Returns

The loss.

4.8.2.7 loss_fn_nograd()

```
double loss_fn_nograd (
            double * yhat,
            double y,
            LOSS loss,
            int size )
```

Calculates the loss of the network (classification only) without values.

Parameters

yhat	The output of the network.
y	The target class.
loss	The loss function.
size	The size of the output.

Returns

The loss.

4.8.2.8 predict()

```
int predict (
            ANN * n,
            double * x,
            int classes )
```

Predicts the class of an input.

Parameters

n	The network.
x	The input.
classes	The number of classes.

Returns

The predicted class.

4.8.2.9 regularization()

```
VALUE * regularization (
            ANN * a,
            REG reg,
            double c )
```

Calculates the regularization term.

Parameters

a	The network.
reg	The regularization type.
c	The regularization coefficient.

4.8.2.10 zero_grad()

```
void zero_grad (
            ANN * a )
```

Sets the gradients of the network to zero.

Parameters

a	The network.

4.9 ann.h

Go to the documentation of this file.

```
00001
00010 #ifndef __ANN_H__
00011 #define __ANN_H__
00012
00013 #include "../utils/grad.h"
00014 #include "layer.h"
00015
00019 typedef enum loss_enum {
00020     MSE,
00021     CROSS_ENTROPY
00022 } LOSS;
00023
00027 typedef struct ann_struct {
00028     int n_layers;
00029     LAYER **layers;
00030 } ANN;
00031
00040 ANN *ann(int num_layers, int *layer_sizes, OPERATION *activations, int num_inputs);
00041
00048 VALUE **ann_forward(ANN *a, VALUE **x);
00049
00056 VALUE *regularization(ANN *a, REG reg, double c);
00057
00064 void ann_descend(ANN *a, double lr, bool momentum);
00065
00070 void free_ann(ANN *a);
00071
00076 void zero_grad(ANN *a);
00077
00086 VALUE *loss_fn(VALUE **yhat, double y, LOSS loss, int size);
00087
00094 double *ann_nograd_forward(ANN *a, double *x);
00095
00104 double loss_fn_nograd(double *yhat, double y, LOSS loss, int size);
00105
00113 int predict(ANN *n, double *x, int classes);
00114
00115 #endif // __ANN_H__
```

4.10 nn/layer.c File Reference

Implementation for a layer of neurons in a feedforward neural network.

```
#include "layer.h"
```

Functions

- LAYER * layer (int num_inputs, int size, OPERATION activation)

  Creates a layer of neurons.
- VALUE ** layer_forward (LAYER *l, VALUE **x)

  Performs a forward pass on the layer.
- double * layer_nograd_forward (LAYER *l, double *x)

  Performs a forward pass on the layer without creating a graph.
- VALUE * layer_regularization (LAYER *l, REG reg, double c)

  Calculates the regularization term for the layer.
- void layer_descend (LAYER *l, double lr, bool momentum)

  Performs a gradient descent step on the layer.
- void free_layer (LAYER *l)

  Frees the memory allocated to the layer.
- void layer_zero_grad (LAYER *l)

  Sets the gradients of the layer to zero.

4.10.1 Detailed Description

Implementation for a layer of neurons in a feedforward neural network.

This contains the implementation for the functions used to create and manipulate a layer of neurons in a feedforward neural network.

Author

Vamsi Deeduvanu (vamsi10010)

4.10.2 Function Documentation

4.10.2.1 free_layer()

```
void free_layer (
            LAYER * l )
```

Frees the memory allocated to the layer.

Parameters

l	The layer.

4.10.2.2 layer()

```
LAYER * layer (
            int num_inputs,
            int size,
            OPERATION activation )
```

Creates a layer of neurons.

Parameters

num_inputs	The number of inputs to each neuron in the layer.
size	The number of neurons in the layer.
activation	The activation function for the layer.

Returns

A pointer to the layer.

4.10.2.3 layer_descend()

```
void layer_descend (
            LAYER * l,
```

```
            double lr,
            bool momentum )
```

Performs a gradient descent step on the layer.

Parameters

l	The layer.
lr	The learning rate.
momentum	Whether to use momentum.

4.10.2.4 layer_forward()

```
VALUE ** layer_forward (
            LAYER * l,
            VALUE ** x )
```

Performs a forward pass on the layer.

Parameters

l	The layer.
x	The input to the layer.

Returns

The output of the layer.

4.10.2.5 layer_nograd_forward()

```
double * layer_nograd_forward (
            LAYER * l,
            double * x )
```

Performs a forward pass on the layer without creating a graph.

Parameters

l	The layer.
x	The input to the layer.

Returns

The output of the layer.

4.10.2.6 layer_regularization()

```
VALUE * layer_regularization (
            LAYER * l,
```

```
            REG reg,
            double c )
```

Calculates the regularization term for the layer.

Parameters

l	The layer.
reg	The regularization type.
c	The regularization coefficient.

Returns

The regularization term.

4.10.2.7 layer_zero_grad()

```
void layer_zero_grad (
            LAYER * l )
```

Sets the gradients of the layer to zero.

Parameters

l	The layer.

4.11 nn/layer.h File Reference

Header file for a layer of neurons in a feedforward neural network.

```
#include "../utils/grad.h"
#include "neuron.h"
```

Classes

- struct layer_struct

  A layer of neurons in a feedforward neural network.

Typedefs

- typedef struct layer_struct **LAYER**

  A layer of neurons in a feedforward neural network.

Functions

- LAYER * layer (int num_inputs, int size, OPERATION activation)

  Creates a layer of neurons.
- VALUE ** layer_forward (LAYER *l, VALUE **x)

  Performs a forward pass on the layer.
- VALUE * layer_regularization (LAYER *l, REG reg, double c)

  Calculates the regularization term for the layer.
- void layer_descend (LAYER *l, double lr, bool momentum)

  Performs a gradient descent step on the layer.
- void free_layer (LAYER *l)

  Frees the memory allocated to the layer.
- void layer_zero_grad (LAYER *l)

  Sets the gradients of the layer to zero.
- double * layer_nograd_forward (LAYER *l, double *x)

  Performs a forward pass on the layer without creating a graph.

4.11.1 Detailed Description

Header file for a layer of neurons in a feedforward neural network.

This contains the prototypes for the functions used to create and manipulate a layer of neurons in a feedforward neural network.

Author

Vamsi Deeduvanu (vamsi10010)

4.11.2 Function Documentation

4.11.2.1 free_layer()

```
void free_layer (
            LAYER * l )
```

Frees the memory allocated to the layer.

Parameters

l	The layer.

4.11.2.2 layer()

```
LAYER * layer (
            int num_inputs,
            int size,
            OPERATION activation )
```

Creates a layer of neurons.

Parameters

num_inputs	The number of inputs to each neuron in the layer.
size	The number of neurons in the layer.
activation	The activation function for the layer.

Returns

A pointer to the layer.

4.11.2.3 layer_descend()

```
void layer_descend (
            LAYER * l,
            double lr,
            bool momentum )
```

Performs a gradient descent step on the layer.

Parameters

l	The layer.
lr	The learning rate.
momentum	Whether to use momentum.

4.11.2.4 layer_forward()

```
VALUE ** layer_forward (
            LAYER * l,
            VALUE ** x )
```

Performs a forward pass on the layer.

Parameters

l	The layer.
x	The input to the layer.

Returns

The output of the layer.

4.11.2.5 layer_nograd_forward()

```
double * layer_nograd_forward (
            LAYER * l,
            double * x )
```

Performs a forward pass on the layer without creating a graph.

Parameters

l	The layer.
x	The input to the layer.

Returns

The output of the layer.

4.11.2.6 layer_regularization()

```
VALUE * layer_regularization (
            LAYER * l,
            REG reg,
            double c )
```

Calculates the regularization term for the layer.

Parameters

l	The layer.
reg	The regularization type.
c	The regularization coefficient.

Returns

The regularization term.

4.11.2.7 layer_zero_grad()

```
void layer_zero_grad (
            LAYER * l )
```

Sets the gradients of the layer to zero.

Parameters

l	The layer.

4.12 layer.h

Go to the documentation of this file.

```
00001
00010 #ifndef __LAYER_H__
00011 #define __LAYER_H__
00012
00013 #include "../utils/grad.h"
```

```
00014 #include "neuron.h"
00015
00019 typedef struct layer_struct {
00020     int size;
00021     NEURON **neurons;
00022     OPERATION activation;
00023 } LAYER;
00024
00032 LAYER *layer(int num_inputs, int size, OPERATION activation);
00033
00040 VALUE **layer_forward(LAYER *l, VALUE **x);
00041
00049 VALUE *layer_regularization(LAYER *l, REG reg, double c);
00050
00057 void layer_descend(LAYER *l, double lr, bool momentum);
00058
00063 void free_layer(LAYER *l);
00064
00069 void layer_zero_grad(LAYER *l);
00070
00077 double *layer_nograd_forward(LAYER *l, double *x);
00078
00079 #endif // __LAYER_H__
```

4.13 nn/neuron.c File Reference

Function implementations for neuron.h.

```
#include "neuron.h"
```

Functions

- NEURON * neuron (int num_inputs)

  Creates a neuron with the given number of inputs.
- void copy_weights (NEURON *n)

  Copies the weights and biases into value structs to construct the computational graph.
- VALUE * neuron_forward (NEURON *n, VALUE **x, OPERATION activation)

  Performs a forward pass on the neuron without building a computational graph.
- double neuron_nograd_forward (NEURON *n, double *x, OPERATION activation)

  Performs a forward pass on the neuron without building a computational graph.
- VALUE * neuron_regularization (NEURON *n, REG reg, double c)

  Calculates the regularization term for the neuron.
- void neuron_descend (NEURON *n, double lr, bool momentum)

  Performs a gradient descent step on the neuron.
- void free_neuron (NEURON *n)

  Frees the memory allocated to the neuron.
- void neuron_zero_grad (NEURON *n)

  Zeroes out the gradients of the neuron.

4.13.1 Detailed Description

Function implementations for neuron.h.

This contains function implementations for creating neurons and performing operations on them.

Author

Vamsi Deeduvanu (vamsi10010)

4.13.2 Function Documentation

4.13.2.1 copy_weights()

```
void copy_weights (
            NEURON * n )
```

Copies the weights and biases into value structs to construct the computational graph.

Parameters

n	The neuron.

4.13.2.2 free_neuron()

```
void free_neuron (
            NEURON * n )
```

Frees the memory allocated to the neuron.

Parameters

n	The neuron.

4.13.2.3 neuron()

```
NEURON * neuron (
            int num_inputs )
```

Creates a neuron with the given number of inputs.

Parameters

num_inputs	The number of inputs to the neuron.

Returns

A pointer to the neuron.

4.13.2.4 neuron_descend()

```
void neuron_descend (
            NEURON * n,
            double lr,
            bool momentum )
```

Performs a gradient descent step on the neuron.

Parameters

n	The neuron.
lr	The learning rate.
momentum	Whether to use momentum in gradient descent.

4.13.2.5 neuron_forward()

```
VALUE * neuron_forward (
            NEURON * n,
            VALUE ** x,
            OPERATION activation )
```

Performs a forward pass on the neuron without building a computational graph.

Performs a forward pass on the neuron.

Parameters

n	The neuron.
x	The inputs to the neuron.
activation	The activation function to use.

4.13.2.6 neuron_nograd_forward()

```
double neuron_nograd_forward (
            NEURON * n,
            double * x,
            OPERATION activation )
```

Performs a forward pass on the neuron without building a computational graph.

Parameters

n	The neuron.
x	The inputs to the neuron.
activation	The activation function to use.

Returns

The output of the neuron.

4.13.2.7 neuron_regularization()

```
VALUE * neuron_regularization (
            NEURON * n,
```

```
        REG reg,
        double c )
```

Calculates the regularization term for the neuron.

Parameters

n	The neuron.
reg	The regularization technique to use.
c	The regularization coefficient.

4.13.2.8 neuron_zero_grad()

```
void neuron_zero_grad (
            NEURON * n )
```

Zeroes out the gradients of the neuron.

Parameters

n	The neuron.

4.14 nn/neuron.h File Reference

Header file for neuron.c.

```
#include "../utils/grad.h"
#include "../utils/normal.h"
```

Classes

- struct neuron_struct

  A neuron in a neural network.

Typedefs

- typedef enum regularization_enum **REG**

  Different regularization techniques.
- typedef struct neuron_struct **NEURON**

  A neuron in a neural network.

Enumerations

- enum regularization_enum { **L1** , **L2** }

  Different regularization techniques.

Functions

- NEURON ∗ neuron (int num_inputs)

  Creates a neuron with the given number of inputs.
- VALUE ∗ neuron_forward (NEURON ∗n, VALUE ∗∗x, OPERATION activation)

  Performs a forward pass on the neuron.
- double neuron_nograd_forward (NEURON ∗n, double ∗x, OPERATION activation)

  Performs a forward pass on the neuron without building a computational graph.
- VALUE ∗ neuron_regularization (NEURON ∗n, REG reg, double c)

  Calculates the regularization term for the neuron.
- void neuron_descend (NEURON ∗n, double lr, bool momentum)

  Performs a gradient descent step on the neuron.
- void free_neuron (NEURON ∗n)

  Frees the memory allocated to the neuron.
- void copy_weights (NEURON ∗n)

  Copies the weights and biases into value structs to construct the computational graph.
- void neuron_zero_grad (NEURON ∗n)

  Zeroes out the gradients of the neuron.

4.14.1 Detailed Description

Header file for neuron.c.

This contains function prototypes and struct definitions to create neurons and perform operations on them.

Author

Vamsi Deeduvanu (vamsi10010)

4.14.2 Function Documentation

4.14.2.1 copy_weights()

```
void copy_weights (
            NEURON * n )
```

Copies the weights and biases into value structs to construct the computational graph.

Parameters

n	The neuron.

4.14.2.2 free_neuron()

```
void free_neuron (
            NEURON * n )
```

Frees the memory allocated to the neuron.

Parameters

n	The neuron.

4.14.2.3 neuron()

```
NEURON * neuron (
            int num_inputs )
```

Creates a neuron with the given number of inputs.

Parameters

num_inputs	The number of inputs to the neuron.

Returns

A pointer to the neuron.

4.14.2.4 neuron_descend()

```
void neuron_descend (
            NEURON * n,
            double lr,
            bool momentum )
```

Performs a gradient descent step on the neuron.

Parameters

n	The neuron.
lr	The learning rate.
momentum	Whether to use momentum in gradient descent.

4.14.2.5 neuron_forward()

```
VALUE * neuron_forward (
            NEURON * n,
            VALUE ** x,
            OPERATION activation )
```

Performs a forward pass on the neuron.

Performs a forward pass on the neuron without building a computational graph.

Parameters

n	The neuron.
x	The inputs to the neuron.
activation	The activation function to use.

Returns

The output of the neuron.

Parameters

n	The neuron.
x	The inputs to the neuron.
activation	The activation function to use.

Performs a forward pass on the neuron.

Parameters

n	The neuron.
x	The inputs to the neuron.
activation	The activation function to use.

4.14.2.6 neuron_nograd_forward()

```
double neuron_nograd_forward (
            NEURON * n,
            double * x,
            OPERATION activation )
```

Performs a forward pass on the neuron without building a computational graph.

Parameters

n	The neuron.
x	The inputs to the neuron.
activation	The activation function to use.

Returns

The output of the neuron.

4.14.2.7 neuron_regularization()

```
VALUE * neuron_regularization (
            NEURON * n,
            REG reg,
            double c )
```

Calculates the regularization term for the neuron.

Parameters

n	The neuron.
reg	The regularization technique to use.
c	The regularization coefficient.

4.14.2.8 neuron_zero_grad()

```
void neuron_zero_grad (
            NEURON * n )
```

Zeroes out the gradients of the neuron.

Parameters

n	The neuron.

4.15 neuron.h

Go to the documentation of this file.

```
00001
00010 #ifndef __NEURON_H__
00011 #define __NEURON_H__
00012
00013 #include "../utils/grad.h"
00014 #include "../utils/normal.h"
00015
00019 typedef enum regularization_enum {
00020     L1,
00021     L2
00022 } REG;
00023
00027 typedef struct neuron_struct {
00028     int num_inputs;
00029     PARAM *params;
00030     VALUE **weights;
00031 } NEURON;
00032
00038 NEURON *neuron(int num_inputs);
00039
00047 VALUE *neuron_forward(NEURON *n, VALUE **x, OPERATION activation);
00048
00057 double neuron_nograd_forward(NEURON *n, double *x, OPERATION activation);
00058
00065 VALUE *neuron_regularization(NEURON *n, REG reg, double c);
00066
00073 void neuron_descend(NEURON *n, double lr, bool momentum);
00074
00079 void free_neuron(NEURON *n);
00080
00088 VALUE *neuron_forward(NEURON *n, VALUE **x, OPERATION activation);
00089
00090
00096 void copy_weights(NEURON *n);
00097
00102 void neuron_zero_grad(NEURON *n);
00103
00104 #endif // __NEURON_H__
```

4.16 utils/grad.c File Reference

Functions for backpropagation.

```
#include "grad.h"
```

Functions

- VALUE * constant (double a)

  Creates a value from a double.
- VALUE * parameter (PARAM *p)

  Creates a value from a parameter.
- VALUE * add (VALUE *a, VALUE *b)

  Adds two values.
- void add_backward (VALUE *v)

  Backward pass for an addition operation.
- VALUE * mul (VALUE *a, VALUE *b)

  Multiplies two values.
- void mul_backward (VALUE *v)

  Backward pass for a multiplication operation.
- VALUE * power (VALUE *a, VALUE *b)

  Raises a value to a power.
- void power_backward (VALUE *v)

  Backward pass for a power operation.
- VALUE * mod (VALUE *a)

  Takes the modulus of a value.
- void mod_backward (VALUE *v)

  Backward pass for a modulus operation.
- VALUE * ex (VALUE *a)

  Takes the exponential (e ^ a) of a value.
- void ex_backward (VALUE *v)

  Backward pass for an exponential operation.
- VALUE * lg (VALUE *a)

  Takes the natural logarithm of a value.
- void lg_backward (VALUE *v)

  Backward pass for a natural logarithm operation.
- VALUE * relu (VALUE *a)

  Takes the rectified linear unit of a value.
- void relu_backward (VALUE *v)

  Backward pass for a RELU operation.
- VALUE * tanhyp (VALUE *a)

  Takes the hyperbolic tangent of a value.
- void tanh_backward (VALUE *v)

  Backward pass for a tanh operation.
- VALUE * sigmoid (VALUE *a)

  Takes the sigmoid of a value.
- void sigmoid_backward (VALUE *v)

  Backward pass for a sigmoid operation.
- VALUE * sub (VALUE *a, VALUE *b)

  Takes the difference of two values.
- VALUE * divide (VALUE *a, VALUE *b)

  Divides of two values.
- VALUE * neg (VALUE *a)

  Takes the negation (-a) of a value.
- VALUE ** softmax (VALUE **x, int size)

  Applies softmax function to an array of values.
- void build_topological_order (VALUE *v, NODE **head)

Sorts a DAG of values into topological order.

- void backward (VALUE *v)

  Backward pass over entire computational graph.

- void free_values (VALUE **v)

  Frees all values under input value in the graph.

- VALUE ** value_array (double *arr, int size)

  Creates an array of values from an array of doubles.

- VALUE * max (VALUE *a, VALUE *b)

  Finds the maximum of two values.

- void argmax (VALUE **args, int size, VALUE **out, int *idx)

  Finds the index of the maximum value in an array of values.

4.16.1 Detailed Description

Functions for backpropagation.

This contains the function implementations to perform bacpropagation on an equation.

Author

Vamsi Deeduvanu (vamsi10010)

4.16.2 Function Documentation

4.16.2.1 add()

```
VALUE * add (
            VALUE * a,
            VALUE * b )
```

Adds two values.

Parameters

a	The first value.
b	The second value.

Returns

The sum of the two values.

4.16.2.2 add_backward()

```
void add_backward (
            VALUE * v )
```

Backward pass for an addition operation.

Parameters

v	The value.

4.16.2.3 argmax()

```
void argmax (
            VALUE ** x,
            int size,
            VALUE ** out,
            int * idx )
```

Finds the index of the maximum value in an array of values.

Parameters

x	The array of values.
size	The size of the array.
out	Pointer to output value.
idx	Pointer to output index.

Returns

The index of the maximum value.

4.16.2.4 backward()

```
void backward (
            VALUE * v )
```

Backward pass over entire computational graph.

Parameters

v	The value.

4.16.2.5 build_topological_order()

```
void build_topological_order (
            VALUE * v,
            NODE ** head )
```

Sorts a DAG of values into topological order.

Parameters

v	The head value.
head	The head of the linked list.

4.16.2.6 constant()

```
VALUE * constant (
            double a )
```

Creates a value from a double.

Parameters

val	The input double.

Returns

The constant value.

4.16.2.7 divide()

```
VALUE * divide (
            VALUE * a,
            VALUE * b )
```

Divides of two values.

Parameters

a	The numerator value.
b	The denominator value.

Returns

The quotient of the two values.

4.16.2.8 ex()

```
VALUE * ex (
            VALUE * a )
```

Takes the exponential (e ^ a) of a value.

Parameters

a	The value.

Returns

The exponential of the value.

4.16.2.9 ex_backward()

```
void ex_backward (
            VALUE * v )
```

Backward pass for an exponential operation.

Parameters

v	The value.

4.16.2.10 free_values()

```
void free_values (
            VALUE ** v )
```

Frees all values under input value in the graph.

Parameters

v	The input value.

4.16.2.11 lg()

```
VALUE * lg (
            VALUE * a )
```

Takes the natural logarithm of a value.

Parameters

a	The value.

Returns

The natural logarithm of the value.

4.16.2.12 lg_backward()

```
void lg_backward (
            VALUE * v )
```

Backward pass for a natural logarithm operation.

Parameters

v	The value.

4.16.2.13 max()

```
VALUE * max (
            VALUE * a,
            VALUE * b )
```

Finds the maximum of two values.

Parameters

a	The first value.
b	The second value.

Returns

The maximum of the two values.

4.16.2.14 mod()

```
VALUE * mod (
            VALUE * a )
```

Takes the modulus of a value.

Parameters

a	The value.

Returns

The modulus of the value.

4.16.2.15 mod_backward()

```
void mod_backward (
            VALUE * v )
```

Backward pass for a modulus operation.

Parameters

v	The value.

4.16.2.16 mul()

```
VALUE * mul (
            VALUE * a,
            VALUE * b )
```

Multiplies two values.

Parameters

a	The first value.
b	The second value.

Returns

The product of the two values.

4.16.2.17 mul_backward()

```
void mul_backward (
            VALUE * v )
```

Backward pass for a multiplication operation.

Parameters

v	The value.

4.16.2.18 neg()

```
VALUE * neg (
            VALUE * a )
```

Takes the negation (-a) of a value.

Parameters

a	The value.

Returns

The negation of the value.

4.16.2.19 parameter()

```
VALUE * parameter (
            PARAM * p )
```

Creates a value from a parameter.

Parameters

val	The input parameter.

Returns

The value.

4.16.2.20 power()

```
VALUE * power (
            VALUE * a,
            VALUE * b )
```

Raises a value to a power.

Parameters

a	The value.
b	The power. Must be a constant value.

Returns

The value raised to the power.

4.16.2.21 power_backward()

```
void power_backward (
            VALUE * v )
```

Backward pass for a power operation.

Parameters

v	The value.

4.16.2.22 relu()

```
VALUE * relu (
            VALUE * a )
```

Takes the rectified linear unit of a value.

Parameters

a	The value.

Returns

The RELU of the value.

4.16.2.23 relu_backward()

```
void relu_backward (
            VALUE * v )
```

Backward pass for a RELU operation.

Parameters

v	The value.

4.16.2.24 sigmoid()

```
VALUE * sigmoid (
            VALUE * a )
```

Takes the sigmoid of a value.

Parameters

a	The value.

Returns

The sigmoid of the value.

4.16.2.25 sigmoid_backward()

```
void sigmoid_backward (
            VALUE * v )
```

Backward pass for a sigmoid operation.

Parameters

v	The value.

4.16.2.26 softmax()

```
VALUE ** softmax (
            VALUE ** x,
            int size )
```

Applies softmax function to an array of values.

Parameters

x	The array of values.
size	The size of the array.

Returns

A new array of values with softmax applied.

4.16.2.27 sub()

```
VALUE * sub (
            VALUE * a,
            VALUE * b )
```

Takes the difference of two values.

Parameters

a	The first value.
b	The second value.

Returns

The difference of the two values.

4.16.2.28 tanh_backward()

```
void tanh_backward (
            VALUE * v )
```

Backward pass for a tanh operation.

Parameters

v	The value.

4.16.2.29 tanhyp()

```
VALUE * tanhyp (
            VALUE * a )
```

Takes the hyperbolic tangent of a value.

Parameters

a	The value.

Returns

The tanh of the value.

4.16.2.30 value_array()

```
VALUE ** value_array (
            double * arr,
            int size )
```

Creates an array of values from an array of doubles.

Parameters

arr	The array of doubles.
size	The size of the array.

Returns

The array of values.

4.17 utils/grad.h File Reference

Function prototypes for grad.c.

```
#include <assert.h>
#include <math.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

Classes

- struct param_struct

  A parameter of a neuron in a neural network.
- struct value_struct

  A singe unit/value in the computational graph during backpropagation.
- struct node_struct

  A node in a linked list.

Typedefs

- typedef enum operation_enum **OPERATION**

  Different operations supported by cgrad.
- typedef struct param_struct **PARAM**

  A parameter of a neuron in a neural network.
- typedef struct value_struct **VALUE**

  A singe unit/value in the computational graph during backpropagation.
- typedef struct node_struct **NODE**

  A node in a linked list.

Enumerations

- enum operation_enum {
  ADD , **MUL** , **POW** , **MOD** ,
  EXP , **LOG** , **RELU** , **TANH** ,
  SIGMOID , **SOFTMAX** , **CONST** , **MAX** }

  Different operations supported by cgrad.

Functions

- VALUE * add (VALUE *a, VALUE *b)

  Adds two values.
- VALUE * mul (VALUE *a, VALUE *b)

  Multiplies two values.
- VALUE * power (VALUE *a, VALUE *b)

  Raises a value to a power.
- VALUE * mod (VALUE *a)

  Takes the modulus of a value.
- VALUE * ex (VALUE *a)

  Takes the exponential (e ^ a) of a value.
- VALUE * lg (VALUE *a)

  Takes the natural logarithm of a value.
- VALUE * relu (VALUE *a)

  Takes the rectified linear unit of a value.
- VALUE * tanhyp (VALUE *a)

  Takes the hyperbolic tangent of a value.
- VALUE * sigmoid (VALUE *a)

  Takes the sigmoid of a value.
- VALUE * sub (VALUE *a, VALUE *b)

  Takes the difference of two values.
- VALUE * divide (VALUE *a, VALUE *b)

  Divides of two values.
- VALUE * neg (VALUE *a)

  Takes the negation (-a) of a value.
- VALUE ** softmax (VALUE **x, int size)

  Applies softmax function to an array of values.
- VALUE * max (VALUE *a, VALUE *b)

  Finds the maximum of two values.
- void add_backward (VALUE *v)

  Backward pass for an addition operation.
- void mul_backward (VALUE *v)

  Backward pass for a multiplication operation.
- void power_backward (VALUE *v)

  Backward pass for a power operation.
- void mod_backward (VALUE *v)

  Backward pass for a modulus operation.
- void ex_backward (VALUE *v)

  Backward pass for an exponential operation.
- void lg_backward (VALUE *v)

  Backward pass for a natural logarithm operation.
- void relu_backward (VALUE *v)

Backward pass for a RELU operation.

- void tanh_backward (VALUE *v)

  Backward pass for a tanh operation.

- void sigmoid_backward (VALUE *v)

  Backward pass for a sigmoid operation.

- void backward (VALUE *v)

  Backward pass over entire computational graph.

- VALUE * constant (double)

  Creates a value from a double.

- VALUE * parameter (PARAM *p)

  Creates a value from a parameter.

- void build_topological_order (VALUE *v, NODE **head)

  Sorts a DAG of values into topological order.

- NODE * build_node (VALUE *v)

  Builds a linked list node from a value.

- VALUE ** value_array (double *arr, int size)

  Creates an array of values from an array of doubles.

- void argmax (VALUE **x, int size, VALUE **out, int *idx)

  Finds the index of the maximum value in an array of values.

- void free_values (VALUE **v)

  Frees all values under input value in the graph.

4.17.1 Detailed Description

Function prototypes for grad.c.

This contains function prototypes for the backpropagation engine of cgrad.

Author

Vamsi Deeduvanu (vamsi10010)

4.17.2 Function Documentation

4.17.2.1 add()

```
VALUE * add (
            VALUE * a,
            VALUE * b )
```

Adds two values.

Parameters

a	The first value.
b	The second value.

Returns

The sum of the two values.

4.17.2.2 add_backward()

```
void add_backward (
            VALUE * v )
```

Backward pass for an addition operation.

Parameters

v	The value.

4.17.2.3 argmax()

```
void argmax (
            VALUE ** x,
            int size,
            VALUE ** out,
            int * idx )
```

Finds the index of the maximum value in an array of values.

Parameters

x	The array of values.
size	The size of the array.
out	Pointer to output value.
idx	Pointer to output index.

Returns

The index of the maximum value.

4.17.2.4 backward()

```
void backward (
            VALUE * v )
```

Backward pass over entire computational graph.

Parameters

v	The value.

4.17.2.5 build_node()

```
NODE * build_node (
            VALUE * v )
```

Builds a linked list node from a value.

Parameters

v	The value.

Returns

The node.

4.17.2.6 build_topological_order()

```
void build_topological_order (
            VALUE * v,
            NODE ** head )
```

Sorts a DAG of values into topological order.

Parameters

v	The head value.
head	The head of the linked list.

4.17.2.7 constant()

```
VALUE * constant (
            double a )
```

Creates a value from a double.

Parameters

val	The input double.

Returns

The constant value.

4.17.2.8 divide()

```
VALUE * divide (
            VALUE * a,
            VALUE * b )
```

Divides of two values.

Parameters

a	The numerator value.
b	The denominator value.

Returns

The quotient of the two values.

4.17.2.9 ex()

```
VALUE * ex (
            VALUE * a )
```

Takes the exponential (e ^ a) of a value.

Parameters

a	The value.

Returns

The exponential of the value.

4.17.2.10 ex_backward()

```
void ex_backward (
            VALUE * v )
```

Backward pass for an exponential operation.

Parameters

v	The value.

4.17.2.11 free_values()

```
void free_values (
            VALUE ** v )
```

Frees all values under input value in the graph.

Parameters

v	The input value.

4.17.2.12 lg()

```
VALUE * lg (
            VALUE * a )
```

Takes the natural logarithm of a value.

Parameters

a	The value.

Returns

The natural logarithm of the value.

4.17.2.13 lg_backward()

```
void lg_backward (
            VALUE * v )
```

Backward pass for a natural logarithm operation.

Parameters

v	The value.

4.17.2.14 max()

```
VALUE * max (
            VALUE * a,
            VALUE * b )
```

Finds the maximum of two values.

Parameters

a	The first value.
b	The second value.

Returns

The maximum of the two values.

4.17.2.15 mod()

```
VALUE * mod (
            VALUE * a )
```

Takes the modulus of a value.

Parameters

a	The value.

Returns

The modulus of the value.

4.17.2.16 mod_backward()

```
void mod_backward (
            VALUE * v )
```

Backward pass for a modulus operation.

Parameters

v	The value.

4.17.2.17 mul()

```
VALUE * mul (
            VALUE * a,
            VALUE * b )
```

Multiplies two values.

Parameters

a	The first value.
b	The second value.

Returns

The product of the two values.

4.17.2.18 mul_backward()

```
void mul_backward (
            VALUE * v )
```

Backward pass for a multiplication operation.

Parameters

v	The value.

4.17.2.19 neg()

```
VALUE * neg (
            VALUE * a )
```

Takes the negation (-a) of a value.

Parameters

a	The value.

Returns

The negation of the value.

4.17.2.20 parameter()

```
VALUE * parameter (
            PARAM * p )
```

Creates a value from a parameter.

Parameters

val	The input parameter.

Returns

The value.

4.17.2.21 power()

```
VALUE * power (
            VALUE * a,
            VALUE * b )
```

Raises a value to a power.

Parameters

a	The value.
b	The power. Must be a constant value.

Returns

The value raised to the power.

4.17.2.22 power_backward()

```
void power_backward (
            VALUE * v )
```

Backward pass for a power operation.

Parameters

v	The value.

4.17.2.23 relu()

```
VALUE * relu (
            VALUE * a )
```

Takes the rectified linear unit of a value.

Parameters

a	The value.

Returns

The RELU of the value.

4.17.2.24 relu_backward()

```
void relu_backward (
            VALUE * v )
```

Backward pass for a RELU operation.

Parameters

v	The value.

4.17.2.25 sigmoid()

```
VALUE * sigmoid (
            VALUE * a )
```

Takes the sigmoid of a value.

Parameters

a	The value.

Returns

The sigmoid of the value.

4.17.2.26 sigmoid_backward()

```
void sigmoid_backward (
            VALUE * v )
```

Backward pass for a sigmoid operation.

Parameters

v	The value.

4.17.2.27 softmax()

```
VALUE ** softmax (
            VALUE ** x,
            int size )
```

Applies softmax function to an array of values.

Parameters

x	The array of values.
size	The size of the array.

Returns

A new array of values with softmax applied.

4.17.2.28 sub()

```
VALUE * sub (
            VALUE * a,
            VALUE * b )
```

Takes the difference of two values.

Parameters

a	The first value.
b	The second value.

Returns

The difference of the two values.

4.17.2.29 tanh_backward()

```
void tanh_backward (
            VALUE * v )
```

Backward pass for a tanh operation.

Parameters

v	The value.

4.17.2.30 tanhyp()

```
VALUE * tanhyp (
            VALUE * a )
```

Takes the hyperbolic tangent of a value.

Parameters

a	The value.

Returns

The tanh of the value.

4.17.2.31 value_array()

```
VALUE ** value_array (
            double * arr,
            int size )
```

Creates an array of values from an array of doubles.

Parameters

arr	The array of doubles.
size	The size of the array.

Returns

The array of values.

4.18 grad.h

Go to the documentation of this file.

```
00001
00010 #ifndef __GRAD_H__
00011 #define __GRAD_H__
00012
00013 #include <assert.h>
00014 #include <math.h>
00015 #include <stdbool.h>
00016 #include <stdio.h>
00017 #include <stdlib.h>
00018 #include <string.h>
00019
00024 typedef enum operation_enum {
00025     ADD,
00026     MUL,
00027     POW,
00028     MOD,
00029     EXP,
00030     LOG,
00031     RELU,
00032     TANH,
00033     SIGMOID,
00034     SOFTMAX,
00035     CONST,
00036     MAX
00037 } OPERATION;
00038
00042 typedef struct param_struct {
00043     double val;
00044     double grad;
00045     double momentum;
00046 } PARAM;
00047
00052 typedef struct value_struct {
00053     double val;
00054     double grad;
00055     void (*backward)(struct value_struct *);
00056     OPERATION op;
00057     struct value_struct *left;
00058     struct value_struct *right;
00059     bool visited;
00060     PARAM *param;
00061 } VALUE;
00062
00067 typedef struct node_struct {
00068     VALUE *value;
00069     struct node_struct *next;
00070 } NODE;
00071
00072 // Operations
00073
00080 VALUE *add(VALUE *a, VALUE *b);
00081
00088 VALUE *mul(VALUE *a, VALUE *b);
00089
00096 VALUE *power(VALUE *a, VALUE *b);
00097
00103 VALUE *mod(VALUE *a);
00104
00110 VALUE *ex(VALUE *a);
00111
00117 VALUE *lg(VALUE *a);
00118
00124 VALUE *relu(VALUE *a);
00125
00131 VALUE *tanhyp(VALUE *a);
00132
00138 VALUE *sigmoid(VALUE *a);
00139
00146 VALUE *sub(VALUE *a, VALUE *b);
00147
00154 VALUE *divide(VALUE *a, VALUE *b);
00155
00161 VALUE *neg(VALUE *a);
00162
00169 VALUE **softmax(VALUE **x, int size);
00170
00177 VALUE *max(VALUE *a, VALUE *b);
00178
00179 // Backward Functions
00180
00185 void add_backward(VALUE *v);
00186
```

```
00191 void mul_backward(VALUE *v);
00192
00197 void power_backward(VALUE *v);
00198
00203 void mod_backward(VALUE *v);
00204
00209 void ex_backward(VALUE *v);
00210
00215 void lg_backward(VALUE *v);
00216
00221 void relu_backward(VALUE *v);
00222
00227 void tanh_backward(VALUE *v);
00228
00233 void sigmoid_backward(VALUE *v);
00234
00239 void backward(VALUE *v);
00240
00241 // Helper Functions
00242
00248 VALUE *constant(double);
00249
00255 VALUE *parameter(PARAM *p);
00256
00262 void build_topological_order(VALUE *v, NODE **head);
00263
00269 NODE *build_node(VALUE *v);
00270
00277 VALUE **value_array(double *arr, int size);
00278
00287 void argmax(VALUE **x, int size, VALUE **out, int *idx);
00288
00289 // Graph Functions
00290
00295 void free_values(VALUE **v);
00296
00297
00298
00299
00300
00301 #endif // __GRAD_H__
```

4.19 utils/normal.c File Reference

Function for generating normally distributed random numbers.

```
#include "normal.h"
```

Macros

- #define **PI** (3.141592653589793)

Functions

- double normal (double mu, double sigma)

  Returns a normally distributed random value with mean mu and standard deviation sigma.

4.19.1 Detailed Description

Function for generating normally distributed random numbers.

This contains function prototype for generating normally distributed random numbers.

Author

Vamsi Deeduvanu (vamsi10010)

4.19.2 Function Documentation

4.19.2.1 normal()

```
double normal (
            double mu,
            double sigma )
```

Returns a normally distributed random value with mean mu and standard deviation sigma.

This function uses an algortihm called `Box-Muller transform`. Use `normal(0, 1)` to generate a standard normal random variable.

Parameters

mu	The mean of the normal distribution.
sigma	The standard deviation of the normal distribution.

4.20 utils/normal.h File Reference

Function prototypes for generating normally distributed random numbers.

```
#include <complex.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

Functions

- double **drand48** (void)
- double normal (double mu, double sigma)

  Returns a normally distributed random value with mean mu and standard deviation sigma.

4.20.1 Detailed Description

Function prototypes for generating normally distributed random numbers.

This contains function prototypes for generating normally distributed random numbers. The function is implemented in normal.c.

Author

Vamsi Deeduvanu (vamsi10010)

4.20.2 Function Documentation

4.20.2.1 normal()

```
double normal (
            double mu,
            double sigma )
```

Returns a normally distributed random value with mean mu and standard deviation sigma.

This function uses an algortihm called `Box-Muller transform`. Use `normal(0, 1)` to generate a standard normal random variable.

Parameters

mu	The mean of the normal distribution.
sigma	The standard deviation of the normal distribution.

4.21 normal.h

Go to the documentation of this file.

```
00001
00011 #ifndef __NORMAL_H__
00012 #define __NORMAL_H__
00013
00014 # include <complex.h>
00015 # include <math.h>
00016 # include <stdio.h>
00017 # include <stdlib.h>
00018 # include <time.h>
00019
00020 double drand48(void);
00021
00032 double normal(double mu, double sigma);
00033
00034 #endif // __NORMAL_H__
```

Index