

## Структуры для определения размерностей:

```
typedef struct shape2
{
    int w;//ширина
    int h;//высота
}shape2;
```

Shape2 это целочисленный двухмерный вектор описывающий ширину и высоту. Используется, например, для описания размера фильтров свертки.

Пример создания:

```
shape2 sh = (shape2){10, 10};
```

```
typedef struct shape3
{
    int w;//ширина
    int h;//высота
    int d;//глубина
} shape3;
```

Shape3 это целочисленный трехмерный вектор описывающий ширину, высоту и глубину трехмерного тензора.

Пример создания:

```
Shape3 sh = (shape3){10, 10, 3};
```

```
typedef struct shape4
{
    int w;//ширина
    int h;//высота
    int d;//глубина
    int b;//количество
}shape4;
```

Shape4 это целочисленный четырехмерный вектор описывающий ширину, высоту, глубину и количество объектов четырехмерного тензора.

Пример создания:

```
Shape4 sh = (shape4){10, 10, 3, 2};
```

## Тензоры:

Общая структура тензора:

```
typedef struct Tensor
```

```

{
    Shape3 s;
    int n;
    float *w;
    float *dw;
    float sumdw;
    void* tData;
} Tensor;

```

Структура tensor описывает гомогенный массив размерностью shape3 s и данными типа float. Сами данные хранятся в динамическом массиве w, общее количество элементов в массиве хранится в n. Также структура хранит массив dw той же размерности. Dw хранит градиенты тензора, а tData это дополнительные данные, необходимые для оптимизации. Объект tData создается при первом вызове функции оптимизации, а формат хранимых данных определяется в зависимости от выбранного алгоритма оптимизации. В дополнительной переменной sumdw сохраняется сумма градиентов тензора для их нормализации.

#### Методы тензора:

```
Tensor Tensor_Create(shape3 s, float c);
```

```
Tensor Tensor_CreateGPU(shape3 s, float c);
```

Создает тензор размерность shape3 s и заполняет элементы массива значением c.

Пример создания:

```
Tensor x = Tensor_Create((shape3){ 100, 100, 3 }, 1.f);
```

```
void Tensor_CopyData(Tensor* dst, Tensor* src);
```

```
void Tensor_CopyDataGPU(Tensor* dst, Tensor* src);
```

Копирует элементы из тензора src в тензор dst.

```
int tldx(shape3 s, int w, int h, int d);
```

Возвращает индекс в одномерном массиве по трехмерной форме, где s- размерность тензора, w,h,d – индексы требуемого элемента по ширине высоте и глубине.

Пример:

```
int l = tldx(x.s, 0, 0, 0);
```

```
void Tensor_Xavier_Rand(float* w, int n);
```

Заполняет элементы массива w с количеством элементов n случайными значениями по непрерывному равномерному распределению.

```
void Tensor_XavierNorm_Rand(float* w, int n);
```

Заполняет элементы массива w с количеством элементов n случайными значениями по непрерывному равномерному распределению.

```
void Tensor_He_Rand(float* w, int n);
```

Заполняет элементы массива w с количеством элементов n случайными значениями по непрерывному равномерному распределению. Случайное число с гауссовым распределением вероятностей (G)

```
void Tensor_Free(Tensor *v);
```

Функция очистки динамической памяти, выделяемой для тензора.

```
float Tensor_Get(Tensor *vol, int w, int h, int d);
```

Возвращает значение, хранящееся в тензоре под индексами w, h, d.

```
void Tensor_Set(Tensor *vol, int w, int h, int d, float v);
```

Устанавливает значение, хранящееся в тензоре под индексами w, h, d в значение v.

```
void Tensor_Copy(Tensor* dst, Tensor *src);
```

Копирует массив значений из тензора src в тензор dst.

```
void Tensor_Copy(Tensor* dst, float *src);
```

Копирует массив значений из массива src в тензор dst.

```
void Tensor_Fill(Tensor* t1, float v);
```

Заполняет значения элементов тензора значением v.

```
float Tensor_WeightedSum(Tensor* t1, Tensor *t2);
```

Возвращает взвешенную сумму элементов тензоров t1 и t2.  $Res += t1.w * t2.w$

```
Tensor* Tensor_Mul(Tensor* t1, Tensor *t2);
```

Умножает тензор t1 на тензор t2.

```
Tensor* Tensor_Add(Tensor* t1, Tensor *t2);
```

Суммирует тензор t1 и тензор t2.

```
shape3 T_Argmax(Tensor *t);
```

Возвращает индексы элемента с максимальным значением в тензоре t.  
Возвращаемые индексы передаются в виде вектора shape3.

```
float T_MinValue(Tensor* t);
```

Возвращает минимальное из всех значений элементов в тензоре t.

```
float T_MaxValue(Tensor* t);
```

Возвращает максимальное из всех значений элементов в тензоре t.

```
float T_Mean(Tensor* t);
```

Возвращает среднее значение всех элементов в тензоре t.

```
cJSON* Tensor_To_JSON(Tensor* v);
```

Конвертирует тензор v в cJSON объект формата json.

```
Tensor* Tensor_From_JSON(cJSON* node);
```

Создает тензор из cJSON объекта node.

```
void Tensor_Load_JSON(Tensor* t, cJSON* node);
```

Загружает данные из cJSON объекта node в тензор t.

```
void* Tensor_Print(Tensor* t);
```

Выводит в консоль структуру и содержимое тензора t.

### Слои:

Данные передаваемые с помощью тензоров обрабатываются и передаются при помощи слоев.

Общая структура слоев:

```
typedef struct Layer
{
    shape3 out_shape;
    shape3 in_shape;
    int n_inputs;
    LayerType type;
    Tensor* input;
    Tensor output;
    void* aData;//additional layer data
} Layer;
```

Основными компонентами слоя являются входной и выходной тензоры. В `input` хранится ссылка на входной тензор слоя. Поле `output` хранит выходные данные слоя. При выполнении операций слоя, данные из `input` обрабатываются функциями соответствующего слоя, и результат операций записывается в `output`. В отличие от `output`, переменная `input` хранит только адрес входного тензора и не инициализируется при создании слоя. Также у слоя есть следующие поля:

- `out_shape` – хранит размерность выходных данных,
- `in_shape` – хранит размерность входных данных,
- `n_inputs` – общее количество элементов, подаваемых на вход,
- `type` – тип слоя,
- `aData` – дополнительный объект данных, которые каждый слой может хранить в зависимости от типа.

В библиотеке доступны следующие типы слоев, представленные в перечислении `LayerType`:

```
typedef enum LayerType {  
    LT_INPUT,  
    LT_DENSE,  
    LT_RELU,  
    LT_SOFTMAX,  
    LT_REGRESSION,  
    LT_CONV,  
    LT_MAXPOOL,  
    LT_MSE,  
    LT_TANHA
```

```
} LayerType;
```

- `LT_INPUT` – входные слои, принимают и передают данные,
- `LT_DENSE` – полносвязные слои нейронных сетей,
- `LT_RELU` – функция активации линейного выпрямления RELU,
- `LT_SOFTMAX` - многопеременная логистическая функция,
- `LT_REGRESSION` – регрессия по одному из элементов выхода,
- `LT_CONV` – сверточные слои,
- `LT_MAXPOOL` - слой дискретизации на основе выборки,
- `LT_MSE` – слои среднеквадратичного отклонения,
- `LT_TANHA` – функция активации гиперболического тангенса.

**Входные слои:**

```
Layer *Input_Create(shape3 out_shape);
```

Функция создает входной слой с выходом размерностью `out_shape` и возвращает его адрес.

```
Tensor *Input_Forward(Layer* l, Tensor* x);
```

Функция прямого прохода входного слоя. Принимает адрес входного слоя `l` и адрес тензора для передачи.

```
void Input_Free(Layer* l);
```

Очистка памяти, выделенной для слоя `l`. Очищаются также и данные самого указателя.

### Полносвязные слои нейронных сетей:

```
typedef struct Dense
```

```
{  
    Tensor *kernels;  
    Tensor biases;  
} Dense;
```

В библиотеке используется модель искусственных нейронов с биосами. Ядра полносвязного слоя и биосы хранятся в структуре `Dense` (`kernels` и `biases`). Переменная `biases` хранит массив биосов со значениями смещений для каждого нейрона в массиве тензоров `kernels`. Объект структуры `Dense` создается при инициализации полносвязного слоя и записывается в поле `aData`.

```
Layer *Dense_Create(int num_neurons, RandType weightInit, float bias, Layer *in);
```

Функция создает полносвязный слой нейронов с выходом размерностью равной количеству нейронов, которое передается в параметре `num_neurons`. При создании слоя также необходимо указать тип инициализации весовых коэффициентов `weightInit` и задать смещение `bias` (обычно = 0). Функция `Dense_Create` возвращает адрес созданного слоя.

Инициализация весовых коэффициентов возможна по одному из следующих типов:

```
typedef enum RandType {  
    R_XAVIER,  
    R_XAVIER_NORM,  
    R_HE,  
    R_ZEROS,  
    R_ONES,  
    R_HALF
```

} RandType;

- R\_XAVIER – вызывает метод Tensor\_Xavier\_Rand для всех kernels в слое,
- R\_XAVIER\_NORM – вызывает метод Tensor\_XavierNorm\_Rand для всех kernels в слое,
- R\_HE – вызывает метод Tensor\_He\_Rand для всех kernels в слое,
- R\_ZEROS – инициализирует все kernels нулевыми значениями,
- R\_ONES - инициализирует все kernels значениями = 1.f
- R\_HALF - инициализирует все kernels значениями = 0.5f

Текущий стандартный подход к инициализации весов слоев и узлов нейронной сети, использующих функцию активации Sigmoid или Tanh, называется инициализацией “Glorot” или “Xavier”. Также для функций активации такого типа подходит модифицированный метод Xavier - тип R\_XAVIER\_NORM. Эти методы инициализации хорошо подходят для случаев, когда функции активации линейны, но плохо работают при нелинейных функциях, таких как Relu. Текущий стандартный подход к инициализации весов слоев и узлов нейронной сети, использующих функцию активации типа Relu, называется инициализацией “he” - тип R\_HE. Для вариантов тестирования различных типов сетей и функций активации, весовые коэффициенты можно инициализировать числами или оставить нулевыми значениями.

Tensor\* Dense\_Forward(Layer\* l);

Функция прямого прохода полносвязного слоя. Принимает адрес полносвязного слоя l и возвращает адрес выходного тензора слоя.

void Dense\_Backward(Layer\* l);

Функция обратного прохода полносвязного слоя. Принимает адрес полносвязного слоя l.

void Dense\_Free(Layer\* l);

Очистка памяти, выделенной для слоя l. Очищаются также и данные самого указателя.

cJSON\* Dense\_To\_JSON(Dense\* d);

Конвертирует данные полносвязного слоя d в cJSON объект формата json.

void Dense\_Load\_JSON(Dense\* d, cJSON\* node);

Загружает данные из cJSON объекта node в данные полносвязного слоя d.

### Сверточные слои нейронных сетей:

```
typedef struct Conv2d
{
    Tensor *kernels;
    Tensor biases;
    shape2 k_size;
    shape2 stride;
    int pad;
}Conv2d;
```

Сверточный слой представляет собой набор карт признаков входных данных. Также, как и с полносвязными слоями, сверточные слои используют сдвиги. Ядра сверточного слоя и биасы хранятся в структуре `Conv2d` (`kernels` и `biases`). Переменная `biases` хранит массив биасов со значениями смещений для каждого ядра свертки в массиве тензоров `kernels`. Также в структуре хранятся:

- размер ядер свертки `k_size` (ширина и высота фильтра),
- шаг смещения `stride` (сдвиг по ширине и высоте),
- величина отступов `pad`.

Объект структуры `Conv2d` создается при инициализации сверточного слоя и записывается в поле `aData`.

```
Layer* Conv2d_Create(int num_kernels, shape2 k_size, shape2 stride, int pad, RandType
weightInit, float bias, Layer* in);
```

Функция создает сверточный слой с выходом размерностью равной:

```
d = num_kernels; //количество ядер свертки
```

```
w = (input.w - k_size.w + pad * 2) / stride.w + 1; //ширина карты признаков
```

```
h = (input.h - k_size.h + pad * 2) / stride.h + 1; //высота карты признаков
```

При создании слоя необходимо указать размер ядер свертки `k_size`, величину смещений `stride` и размер отступов `pad`.

Также, при создании слоя необходимо указать тип инициализации весовых коэффициентов `weightInit` и задать смещение `bias` (обычно = 0). Функция `Conv2d_Create` возвращает адрес созданного слоя.

Инициализация весовых коэффициентов возможна по одному из следующих типов.

```
Tensor* Conv2d_Forward(Layer* l);
```



Функция прямого прохода сверточного слоя. Принимает адрес сверточного слоя `l` и возвращает адрес выходного тензора слоя.

```
void Conv2d_Free(Layer* l);
```

Очистка памяти, выделенной для слоя `l`. Очищаются также и данные самого указателя.

```
void Conv2d_Backward(Layer* l);
```

Функция обратного прохода сверточного слоя. Принимает адрес сверточного слоя `l`.

```
cJSON* Conv2d_To_JSON(Conv2d* d);
```

Конвертирует данные сверточного слоя `d` в cJSON объект формата json.

```
void Conv2d_Load_JSON(Conv2d* d, cJSON* node);
```

Загружает данные из cJSON объекта `node` в данные сверточного слоя `d`.

#### **Слои дискретизации на основе выборки:**

```
typedef struct MaxPool2d
```

```
{  
    shape2 k_size;  
    shape2 stride;  
    int pad;  
}MaxPool2d;
```

Операция выборки это процесс сжатия (уменьшения размеров) входного тензора путём объединения значений блоков, например, за счет выбора максимальных или средних значений из окна выборки. Размер окна хранится в переменной `k_size`. Шаг смещения определяет количество элементов, на которые необходимо сдвигать прямоугольную сетку при выполнении операции выборки. Размер шага хранится в переменной `stride`.

В структуре `MaxPool2d` хранятся:

- размер окна `k_size` (ширина и высота окна),
- шаг смещения `stride` (сдвиг по ширине и высоте),
- величина отступов `pad`.

Объект структуры `MaxPool2d` инициализируется при вызове функции `MaxPool2d_Create` и записывается в поле `aData` слоя.

```
Layer* MaxPool2d_Create(shape2 k_size, shape2 stride, int pad, Layer* in);
```

Функция создает MaxPool слои с выходом размерностью равной:

```
d = input.d; //глубина выходного тензора
```

```
w = (input.w - k_size.w + pad * 2) / stride.w + 1; //ширина выходного тензора
```

```
h = (input.h - k_size.h + pad * 2) / stride.h + 1; //высота выходного тензора
```

При создании слоя необходимо указать размер окна `k_size`, величину смещений `stride` и размер отступов `pad`. Функция `MaxPool2d_Create` возвращает адрес созданного слоя.

```
Tensor* MaxPool2d_Forward(Layer* l);
```

Функция прямого прохода MaxPool слоя. Принимает адрес MaxPool слоя `l` и возвращает адрес выходного тензора слоя.

```
void MaxPool2d_Backward(Layer* l);
```

Функция обратного прохода MaxPool слоя. Принимает адрес MaxPool слоя `l`.

```
void MaxPool2d_Free(Layer* l);
```

Очистка памяти, выделенной для слоя `l`. Очищаются также и данные самого указателя.

### **Слой линейного выпрямления RELU.**

Слой типа RELU выполняет операции нелинейной функции активации над входным тензором. Она преобразует входное значение в значение от 0 до положительной бесконечности. Если входное значение меньше или равно нулю, то ReLU выдает ноль, в противном случае - входное значение. Также в библиотеке доступна реализация линейного выпрямления с «утечкой», которая добавляет небольшую константу к входному значению, позволяя избежать проблемы, когда градиенты равны 0.

```
Layer* Relu_Create(Layer* in);
```

Функция создает Слой линейного выпрямления Relu для входного слоя `in`. Функция `Relu_Create` возвращает адрес созданного слоя.

```
Tensor* Relu_Forward(Layer* l);
```

Функция прямого прохода Relu слоя. Принимает адрес Relu слоя `l` и возвращает адрес выходного тензора слоя.

```
void Relu_Backward(Layer* l);
```

Функция обратного прохода Relu слоя. Принимает адрес Relu слоя `l`.

### **Слой операции гиперболического тангенса.**

Слой типа TanhA выполняет операции функция активации гиперболического тангенса над входным тензором. Она преобразует входное значение в значение в диапазоне от -1 до 1.

```
Layer *TanhA_Create(Layer *in);
```

Функция создает слой TanhA для входного слоя in. Функция TanhA\_Create возвращает адрес созданного слоя.

```
Tensor* TanhA_Forward(Layer* l);
```

Функция прямого прохода TanhA слоя. Принимает адрес TanhA слоя l и возвращает адрес выходного тензора слоя.

```
void TanhA_Backward(Layer* l);
```

Функция обратного прохода TanhA слоя. Принимает адрес TanhA слоя l.

### **Слой операции многопеременной логистической функции.**

Слой типа SoftmaxA выполняет операции функция активации и преобразует входной тензор в тензор той же размерности, где каждый элемент полученного тензора представлен вещественным числом в интервале от 0 до 1 и сумма координат равна 1.

```
Layer * SoftmaxA_Create(Layer *in);
```

Функция создает слой SoftmaxA для входного слоя in. Функция SoftmaxA\_Create возвращает адрес созданного слоя.

```
Tensor* SoftmaxA_Forward(Layer* l);
```

Функция прямого прохода SoftmaxA слоя. Принимает адрес SoftmaxA слоя l и возвращает адрес выходного тензора слоя.

```
void SoftmaxA_Backward(Layer* l);
```

Функция обратного прохода SoftmaxA слоя. Принимает адрес SoftmaxA слоя l.

### **Слои для вычисления ошибки.**

```
typedef struct LData  
{  
    int step;  
    float loss;  
}LData;
```

Слои для вычисления ошибки принимают в качестве параметра тензоры с ожидаемыми выходами сети или с параметрами для вычисления ошибки. Информация о текущем состоянии оптимизации записывается в процессе обучения в структуре LData, которая хранится у слоя в поле aData. LData содержит:

- текущий шаг оптимизации step,
- текущее значение ошибки loss.

#### **Слой оценки среднеквадратического отклонения.**

Слой среднеквадратичной ошибки MSE вычисляет среднее арифметическое квадратов разностей между предсказанными и реальными значениями модели.

```
Layer *MSE_Create(Layer* in);
```

Функция создает слой MSE для входного слоя in. Функция MSE\_Create возвращает адрес созданного слоя.

```
Tensor *MSE_Forward(Layer* l);
```

Функция прямого прохода MSE слоя. Принимает адрес MSE слоя l и возвращает адрес выходного тензора слоя.

```
void MSE_Backward(Layer* l, Tensor* y_true);
```

Функция обратного прохода MSE слоя. Принимает адрес MSE слоя l. Вычисляет градиенты между выходом сети и ожидаемым выходом сети y\_true. После выполнения записывает значение ошибки в объекте LData.

#### **Слой функции ошибки перекрестной энтропии.**

```
typedef struct Softmax
```

```
{
```

```
    float* sums;
```

```
}Softmax;
```

Применяет многопеременную логистическую функцию активации softmax и вычисляет функцию потерь перекрестной энтропии. В структуре Softmax хранятся дополнительные данные для вычисления нормализованного выхода слоя sums.

```
Layer *Softmax_Create(Layer *in);
```

Функция создает слой Softmax для входного слоя in. Функция Softmax\_Create возвращает адрес созданного слоя.

```
Tensor *Softmax_Forward(Layer* l);
```

Функция прямого прохода Softmax слоя. Принимает адрес Softmax слоя l и возвращает адрес выходного тензора слоя.

```
void Softmax_Backward(Layer* l, Tensor* y);
```

Функция обратного прохода Softmax слоя. Принимает адрес Softmax слоя l. Вычисляет градиенты между выходом сети и ожидаемым выходом сети y. После выполнения записывает значение ошибки в объекте LData.

### Создание модели.

Данные модели хранятся в структуре Model.

```
typedef struct _Model
{
    Layer** Layers;
    int n_layers;
    Tensor* (*ModelForward) (struct _Model * m, Tensor* x);
    void (*ModelBackward) (struct _Model * m, Tensor* y);
}Model;
```

Структура Model состоит из следующих полей:

- массив слоев модели Layers – хранит указатели на созданные слои,
- количество слоев в модели layers,
- функция вызова прямого прохода слоев модели ModelForward,
- функция обратного прохода слоев модели ModelBackward

Код для работы с моделями содержит следующие функции:

```
Model Model_Create();
```

Создает объект Model с параметрами по умолчанию. Необходимо вызвать данную функцию при создании модели.

Пример:

```
Model m = Model_Create();
```

```
Layer* Model_AddLayer(Model *m, Layer* l);
```

Добавляет новый слой l в массив Layers модели m и возвращает его адрес.

```
void Backward_Layer (Layer* l, Tensor* y);
```

Общая функция для вызова операции обратного прохода слоя. Вызывает backward функцию соответствующего слоя в зависимости от его типа.

```
Tensor* Forward_Layer(Layer* l, Tensor* x);
```

Общая функция для вызова операции прямого прохода слоя. Вызывает forward функцию соответствующего слоя в зависимости от его типа.

```
Tensor* Seq_Forward(Model* m, Tensor* x);
```

Функция прямого прохода всех слов в модели m для тензоров x. Данная функция по умолчанию записывается в поле ModelForward при создании модели.

```
void Seq_Backward(Model* m, Tensor* y);
```

Функция обратного прохода всех слов в модели m для ожидаемых тензоров y. Данная функция по умолчанию записывается в поле ModelBackward при создании модели.

Пример создания модели:

```
shape input = { 227,227,3 };
```

```
Model m = Model_Create();
```

```
Layer* l = Model_AddLayer(&m, Input_Create(input));
```

```
l = Model_AddLayer(&m, Conv2d_Create(16, { 11,11 }, { 4,4 }, 0, R_HE, 0, l));
```

```
l = Model_AddLayer(&m, Relu_Create(l));
```

```
l = Model_AddLayer(&m, MaxPool2d_Create({ 5,5 }, { 2,2 }, 0, l));
```

```
l = Model_AddLayer(&m, Conv2d_Create(16, { 3,3 }, { 1,1 }, 0, R_HE, 0, l));
```

```
l = Model_AddLayer(&m, Relu_Create(l));
```

```
l = Model_AddLayer(&m, MaxPool2d_Create({ 3,3 }, { 1,1 }, 0, l));
```

```
l = Model_AddLayer(&m, Dense_Create(32, R_HE, 0, l));
```

```
l = Model_AddLayer(&m, Relu_Create(l));
```

```
l = Model_AddLayer(&m, Dense_Create(3, R_XAVIER, 0, l));
```

```
Layer* out = Model_AddLayer(&m, MSE_Create(l));
```

Результат создания модели (вывод с консоли):

Input, output shape: [227, 227, 3]

Conv2d, output shape: [55, 55, 16] pad: 0

Relu, output shape: [55, 55, 16]

MaxPool2d output shape: [26, 26, 16]

Conv2d, output shape: [24, 24, 16] pad: 0

Relu, output shape: [24, 24, 16]

MaxPool2d output shape: [22, 22, 16]

Dense, output shape: [1, 1, 32]

Relu, output shape: [1, 1, 32]

Dense, output shape: [1, 1, 3]

Mse, output shape: [1, 1, 3]

Для вызова прямого прохода сети можно вызвать функцию прямого прохода:

```
Tensor* y = m.NetForward(&m, &x);
```

Вывод тензора y на консоль (результат после обучения):

```
[[[1.00414896]] [[-0.00698878]] [[-0.00264287]]]
```

В данном случае y получает адрес выходных тензоров модели с результатами работы модели. Очищать память по этому указателю после выполнения операции не нужно, память будет очищена при очищении модели.

### Оптимизация:

После того как модель создана ее можно обучить на определенном наборе данных. Обучение осуществляется при помощи оптимизации функции ошибки между ожидаемым выходом модели и ее текущим значением. В библиотеке доступны следующие алгоритмы оптимизации, передаваемые в перечислении OptMethod:

```
typedef enum OptMethod {
```

```
    ADAGRAD,  
    RMSPROP,  
    ADAM,  
    ADAN,  
    NRMSPROP,  
    SGD
```

```
} OptMethod;
```

- ADAGRAD – метод адаптивного градиентного спуска (adaptive gradient algorithm),
- RMSPROP – метод модифицированного адаптивного градиентного спуска (root mean square propagation),
- ADAM – метод адаптивной оценки момента (Adaptive Moment Estimation),
- ADAN - адаптивный алгоритм импульса Нестерова (Adaptive Nesterov Momentum Algorithm),
- NRMSPROP – алгоритм Нестерова, метод накопления импульса (Nesterov Accelerated Gradient),
- SGD – метод простого стохастического градиентного спуска.

Параметры для процесса оптимизации хранятся в структуре OptParams:

```
typedef struct OptParams
```

```
{
```

```
    float learning_rate;  
    OptMethod method;  
    float eps;
```

```

    int counter;
    float b1, b2, b3;
    float decay;
    float b;
    float clip;
}OptParams;

```

- learning\_rate – параметр скорости оптимизации,
- method – метод алгоритма оптимизации,
- eps – небольшое число (обычно 1e-8), служит для предотвращения деления на 0 в некоторых алгоритмах оптимизации,
- counter – счетчик итераций обучения,
- b1, b2, b3, decay – дополнительные коэффициенты, используемые в некоторых алгоритмах оптимизации,
- b – значение импульса, используется в некоторых алгоритмах оптимизации,
- clip – значение отсечки для предотвращения переполнения градиентов в некоторых алгоритмах оптимизации.

Каждый из методов, кроме метода простого SGD, использует дополнительные массивы данных при обработке градиентов. Эти массивы хранятся в дополнительных структурах, согласно методу оптимизации, и записываются в поле tData тензоров, веса которых необходимо оптимизировать. Структуры для хранения данных оптимизации:

```

typedef struct adanTData
{
    float* mk;
    float* vk;
    float* nk;
    float* gprev;
}adanTData;

```

adanTData – используется методом Adan.

```

typedef struct adamTData
{
    float* mt;
    float* vt;
}adamTData;

```

adamTData – используется методом Adam.

```

typedef struct momentumTData
{
    float* vk;

```



```
}momentumTData;
```

momentumTData – используется адаптивными методами оптимизации, такими как Adagrad, RMSProp и NRMSProp.

Список функций:

```
void CreateAdanData(Tensor* t);
```

Функция создает дополнительные данные оптимизации для тензора t при использовании метода оптимизации Adan.

```
void CreateAdamData(Tensor* t);
```

Функция создает дополнительные данные оптимизации для тензора t при использовании метода оптимизации Adam.

```
void CreateMomentumData(Tensor* t);
```

Функция создает дополнительные данные оптимизации для тензора t при использовании адаптивных методов оптимизации Adagrad, RMSProp и NRMSProp.

```
void AdanOpt(Tensor* v, OptParams* par);
```

Функция применяет градиенты к весовым коэффициентам тензора v, с помощью параметров оптимизации par по методу оптимизации Adan.

```
void AdamOpt(Tensor* v, OptParams* par);
```

Функция применяет градиенты к весовым коэффициентам тензора v, с помощью параметров оптимизации par по методу оптимизации Adam.

```
void AdagradOpt(Tensor* v, OptParams* par);
```

Функция применяет градиенты к весовым коэффициентам тензора v, с помощью параметров оптимизации par по методу оптимизации Adagrad.

```
void RMSPropOpt(Tensor* v, OptParams* par);
```

Функция применяет градиенты к весовым коэффициентам тензора v, с помощью параметров оптимизации par по методу оптимизации RMSProp.

```
void NRMSPropOpt(Tensor* v, OptParams* par);
```

Функция применяет градиенты к весовым коэффициентам тензора v, с помощью параметров оптимизации par по методу оптимизации NRMSProp.

```
void SGDOpt(Tensor* v, OptParams* par);
```

Функция применяет градиенты к весовым коэффициентам тензора *v*, с помощью параметров оптимизации *par* по методу оптимизации простого градиентного спуска.

```
OptParams OptParams_Create();
```

Функция создает и возвращает объект *OptParams* с параметрами оптимизации по умолчанию.

```
void Optimize(Model*n, OptParams *par, Tensor *x, Tensor *y);
```

Функция оптимизации модели *n* с параметрами оптимизации *par* для входных данных *x* и ожидаемым выходом модели *y*.

```
void Change_Grad(OptParams* par, Tensor* v, bool norm);
```

Вспомогательная функция для применения градиентов к весовым коэффициентам тензора *v*, с помощью параметров оптимизации *par*. Дополнительный параметр *norm* определяет необходимость применения нормализации к градиентам. Функция выбирает метод согласно тому, что указан в объекте *par*, и вызывает соответствующую функцию оптимизации.

Пример применения шага оптимизации:

```
OptParams par = OptParams_Create();  
par.learning_rate = 0.001f;  
par.method = ADAN;  
Optimize(&m, &par, &x, &y);
```

### Оптимизация на GPU.

Для того, чтобы ускорить требовательные к вычислительной мощности процессы, в библиотеке была реализована возможность создания и обучения моделей на графических процессорах с поддержкой технологии CUDA. Поддерживаются устройства с версией архитектуры *sm\_60* (Pascal) и выше.

Поддержка графического процессора позволяет быстрее обучить модель глубокой сети и затем перенести обученную модель на агента. Либо можно дообучать модель в процессе работы агента и передавать ему текущее состояние нейросети.

Создание и оптимизация моделей на GPU происходит по таким же структурам что и для CPU, только к имени вызываемой функции в конце добавляется "GPU". Например, функция создания тензора на GPU выглядит следующим образом:

```
Tensor x = Tensor_CreateGPU((shape3){ 128, 128,3 }, 0.f);
```

Данный код создаст тензор размерностью 128x128x3 в памяти графического процессора. Важно отметить, что эти данные доступны только для обработки на GPU и не

могут быть напрямую прочитаны или записаны из основного кода, работающего с CPU. Работа с данными, расположенными на GPU, должны осуществляться с помощью соответствующих функций и затем копироваться в оперативную память.

Пример создания модели на GPU:

```
Model m = Net_CreateGPU();
Layer *l = Net_AddLayer(&m, Input_CreateGPU({ 128,128,3 }));
l = Net_AddLayer(&m, Dense_CreateGPU(10, R_XAVIER, 0, l));
l = Net_AddLayer(&m, Dense_CreateGPU(10, R_XAVIER, 0, l));
l = Net_AddLayer(&m, Dense_CreateGPU(2, R_XAVIER, 0, l));
```

Вывод на консоль результата создания модели:

```
Input GPU, output shape: [128, 128, 3]
Dense GPU, output shape: [1, 1, 10]
Dense GPU, output shape: [1, 1, 10]
Dense GPU, output shape: [1, 1, 2]
```

Для оптимизации созданной модели на GPU используется аналогичная CPU версии функция:

```
OptParams par = OptParams_Create();
par.learning_rate = 0.001f;
par.method = ADAN;
OptimizeGPU(&m, &par, &x, &y);
```

### **Дополнительные структуры данных и функции:**

#### **Динамический список:**

```
#define getLElem(E, L, I) ((E*)L.data[I].e)
```

Вспомогательная функция, возвращает прямую ссылку на объект с индексом I в динамическом списке L и типом объекта E.

```
#define getLEInfo(E, L, I) ((E*)L.data[I].i)
```

Вспомогательная функция, возвращает прямую ссылку на дополнительные данные объекта с индексом I в динамическом списке L и типом объекта E.

```
#define LElem(T, E) ((T*)E->e)
```

Вспомогательная функция, возвращает прямую ссылку на объект E и типом объекта T в контейнере типа dElem.

```
typedef struct dElem
```

```
{
    void* e;//element host
    void* i;//element info
}dElem;
```

Структура элементов динамического списка, содержит ссылку на объект *e* и дополнительные данные объекта *i*.

```
typedef struct dList
{
    int length;
    dElem* data;
}dList;
```

Динамически расширяемый список, который может хранить указатели на объекты в контейнерах *dElem*. Структура *dList* содержит: текущую длину динамического списка *length*, и данные списка *data*.

Функции для работы с динамическими списками:

```
dList dList_create();
```

Функция создает и инициализирует пустой динамический список.

```
void dList_realloc(dList* d);//add new elem
```

Функция расширяет выделенную для данных списка память, для того, чтобы можно было добавить новый элемент.

```
void* dList_push(dList* d, void* t);//add and assign
```

Функция добавляет новый элемент *t* в динамический список *d* и возвращает указатель на этот элемент.

```
void dList_free(dList* d);//clear list
```

Функция очищает память, выделенную для динамического списка. Память по указателям на элементы, хранимые в списке, не очищается, очищаются только контейнеры.

#### **Вспомогательные математические функции:**

```
float DegToRad(float deg);
```

Конвертирует угол из градусов *deg* в радианы.

```
float RadToDeg(float rad);
```

Конвертирует угол из радианов *rad* в градусы.

```
float Lerp(float a, float b, float t);
```

Функция линейной интерполяции. Возвращает число на отрезке в диапазоне от a до b, в соответствии со значением параметра t [0,1], задающим положение.

```
float InvLerp(float a, float b, float t);
```

Функция обратная функции lerp, возвращает число [0,1] определяющее положение числа t на отрезке [a, b].

```
float rngFloat();
```

Возвращает случайное число в интервале [0,1]

```
int rngInt(int min, int max);
```

Возвращает целое случайное число в диапазоне от минимального значения min до максимального значения max.

```
float rngNormal();
```

Возвращает случайное число с нормальным распределением вероятностей.

```
void InsertionSort(float* values, int n);
```

Функция выполняет простую сортировку вставками над массивом values с длиной n.

```
float Mean(float* items, int n);
```

Функция возвращает среднее значение элементов массива items с длиной n.

### **Структура кватерниона:**

Кватернионы - система гиперкомплексных чисел, образующая векторное пространство размерностью четыре над полем вещественных чисел. Кватернионы удобны для описания изометрий трёх и четырёхмерного евклидовых пространств и поэтому получили широкое распространение в механике. Кватернионы удобны при работе с поворотами в трехмерном пространстве.

```
typedef struct TQuaternion  
{  
    float x;  
    float y;  
    float z;  
    float w;  
}TQuaternion;
```

Структура четырехмерного вектора  $x, y, z, w$ .

```
TQuaternion TQuaternion_Create(float x, float y, float z, float w);
```

Создает кватернион по значениям  $x, y, z, w$ .

```
TQuaternion TQuaternion_CreateV(TVec3 v, float w);
```

Создает кватернион из трехмерного вектора  $v$  и компоненты  $w$ .

```
TQuaternion TQuaternion_FromVec3(TVec3 axis, float angleRadian);
```

Создает кватернион из трехмерного вектора  $v$  и угла поворота в радианах.

```
TQuaternion TQuaternion_Norm(TQuaternion v);
```

Функция нормализации кватерниона  $v$ .

```
TQuaternion TQuaternion_Conjugate(TQuaternion v);
```

Функция сопряжения кватерниона. Возвращает кватернион с теми же значениями, но с измененным знаком мнимой части

```
TQuaternion TQuaternion_Mul(TQuaternion q1, TQuaternion q2);
```

Функция возвращает результат (кватернион) умножения кватернионов  $q1$  и  $q2$ .

```
TQuaternion TQuaternion_Euler(float x, float y, float z);
```

Функция возвращает кватернион, который поворачивает на  $x$  градусов вокруг оси  $x$ ,  $y$  градусов вокруг оси  $y$  и  $z$  градусов вокруг оси  $z$ .

```
TVec3 TQuaternion_Rotate(TQuaternion q, TVec3 pt);
```

Функция поворачивает точку  $pt$  в соответствии с кватернионом  $q$  и возвращает результат поворота.

**Двухмерный вещественный вектор:**

```
typedef struct TVec2
```

```
{
```

```
    float x;
```

```
    float y;
```

```
}TVec2;
```

Структура двухмерного вещественного вектора со значениями  $x, y$ .

```
TVec2 TVec2_Create(float x, float y);
```

Функция создает двухмерный вектор со значениями  $x, y$ .

```
TVec2 TVec2_Create2(float all);
```

Функция создает двухмерный вектор со значениями  $x = all$ ,  $y = all$ .

```
TVec2 TVec2_Mul(TVec2 v, float d);
```

Функция возвращает результат умножения вектора  $v$  на число  $d$ .

```
TVec2 TVec2_Div(TVec2 v, float d);
```

Функция возвращает результат деления вектора  $v$  на число  $d$ .

```
TVec2 TVec2_Sub(TVec2 v1, TVec2 v2);
```

Функция возвращает результат вычитания вектора  $v2$  из вектора  $v1$ .

```
TVec2 TVec2_Add(TVec2 v1, TVec2 v2);
```

Функция возвращает результат добавления вектора  $v2$  к вектору  $v1$ .

```
TVec2 TVec2_Norm(TVec2 v);
```

Функция нормализации вектора  $v$ . Возвращает нормализованный вектор.

```
TVec2 TVec2_Dir(TVec2 org, TVec2 dest); //direction vector
```

Функция возвращает нормализованный вектор направления  $[0,1]$  от точки  $org$  к точке  $dest$ .

```
float TVec2_Length(TVec2 v);
```

Функция возвращает длину вектора  $v$ .

```
float TVec2_Dot(TVec2 v1, TVec2 v2);
```

Функция возвращает скалярное произведение векторов  $v1$  и  $v2$ .

```
float TVec2_AngleDeg(TVec2 v1, TVec2 v2);
```

Функция возвращает угол в градусах между векторами  $v1$  и  $v2$ .

**Трехмерный вещественный вектор:**

```
typedef struct TVec3
```

```
{  
    float x;  
    float y;  
    float z;
```

```
}TVec3;
```

Структура трехмерного вещественного вектора со значениями  $x$ ,  $y$ ,  $z$ .

```
TVec3 TVec3_Create(float x, float y, float z);
```

Функция создает трехмерный вектор со значениями  $x$ ,  $y$ ,  $z$ .

```
TVec3 TVec3_Create2(float all);
```

Функция создает трехмерный вектор со значениями  $x = all$ ,  $y = all$ ,  $z = all$ .

```
TVec3 TVec3_Mul(TVec3 v, float d);
```

Функция возвращает результат умножения вектора  $v$  на число  $d$ .

```
TVec3 TVec3_Div(TVec3 v, float d);
```

Функция возвращает результат деления вектора  $v$  на число  $d$ .

```
TVec3 TVec3_Sub(TVec3 v1, TVec3 v2);
```

Функция возвращает результат вычитания вектора  $v2$  из вектора  $v1$ .

```
TVec3 TVec3_Add(TVec3 v1, TVec3 v2);
```

Функция возвращает результат добавления вектора  $v2$  к вектору  $v1$ .

```
TVec3 TVec3_Norm(TVec3 v);
```

Функция нормализации вектора  $v$ . Возвращает нормализованный вектор.

```
TVec3 TVec3_Cross(TVec3 v1, TVec3 v2);
```

Функция возвращает векторное произведение векторов  $v1$  и  $v2$ .

```
TVec3 TVec3_Dir(TVec3 org, TVec3 dest); //direction vector
```

Функция возвращает нормализованный вектор направления  $[0,1]$  от точки  $org$  к точке  $dest$ .

```
TVec3 TVec3_Middle(TVec3 org, TVec3 dest);
```

Функция возвращает точку на середине отрезка между  $org$  и  $dest$ .

```
float TVec3_Length(TVec3 v);
```

Функция возвращает длину вектора  $v$ .

```
float TVec3_Dot(TVec3 v1, TVec3 v2);
```

Функция возвращает скалярное произведение векторов  $v1$  и  $v2$ .



```
float TVec3_AngleRad(TVec3 v1, TVec3 v2);
```

Функция возвращает угол в радианах между векторами v1 и v2.

**Четырехмерный вещественный вектор:**

```
typedef struct TVec4
```

```
{
```

```
    float x;
```

```
    float y;
```

```
    float z;
```

```
    float w;
```

```
}TVec4;
```

Структура четырехмерного вещественного вектора со значениями x, y, z, w.

```
TVec4 TVec4_Create(float x, float y, float z, float w);
```

Функция создает четырехмерный вектор со значениями x, y, z, w.

```
TVec4 TVec4_Create3(float x, float y, float z);
```

Функция создает четырехмерный вектор со значениями x, y, z, w=1.f.

```
TVec4 TVec4_Create1(float all);
```

Функция создает четырехмерный вектор со значениями x=all, y=all, z=all, w=all.

```
TVec4 TVec4_Mul(TVec4 v, float d);
```

Функция возвращает результат умножения вектора v на число d.

```
TVec4 TVec4_Div(TVec4 v, float d);
```

Функция возвращает результат деления вектора v на число d.

```
TVec4 TVec4_Sub(TVec4 v1, TVec4 v2);
```

Функция возвращает результат вычитания вектора v2 из вектора v1.

```
TVec4 TVec4_Norm(TVec4 v);
```

Функция нормализации вектора v. Возвращает нормализованный вектор.

```
float TVec4_Dot(TVec4 v1, TVec4 v2);
```

Функция возвращает скалярное произведение векторов v1 и v2.

**Структура луча:**

Луч - это часть линии, которая имеет фиксированную начальную точку, но не имеет конечной точки. Он может простирается бесконечно в одном направлении. Поскольку у луча нет конца, мы не можем измерить его длину.

```
typedef struct TRay
{
    TVec3 org;
    TVec3 dir;
}TRay;
```

Структура луча включает: точку начала луча org, вектор направления луча dir.

```
TVec3 TRay_OnRay(TRay r, float dist);
```

Функция возвращает точку на луче r, расположенную на дистанции dist от начала луча.