ceras

Generated by Doxygen 1.9.2

Chapter 1

Namespace Index

1.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 2

Concept Index

2.1 Concepts

Here is a list of all concepts with brief descriptions:

Chapter 3

Hierarchical Index

3.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 4

Class Index

4.1 Class List

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

Chapter 5

File Index

5.1 File List

Here is a list of all files with brief descriptions:

Chapter 6

Namespace Documentation

6.1 ceras Namespace Reference

Namespaces

- namespace ceras_private
- namespace dataset

Classes

- struct adadelta
- struct adagrad
- struct adam
- struct binary_operator

  A binary operator is composed of a.) a left-side input expression, b.) a right-side input expression, c.) a forward action and d.) a backward action.
- struct compiled_model
- struct complex
- struct constant

  Creates a constant expression from a tensor-like object.
- struct gradient_descent
- struct identity_output_shape_calculator

  The default identity output shape calculator for unary/binary operators. Should be overrided for some special operators.
- struct is_binary_operator
- struct is_binary_operator< binary_operator< Lhs_Operator, Rhs_Operator, Forward_Action, Backward_Action, Output_Shape
- struct is_complex
- struct is_complex< complex< Real_Ex, Imag_Ex > >
- struct is_constant
- struct is_constant< constant< Tsor > >
- struct is_place_holder
- struct is_place_holder< place_holder< Tsor > >
- struct is_tensor
- struct is_tensor< tensor< T, A > >
- struct is_unary_operator
- struct is_unary_operator< unary_operator< Operator, Forward_Action, Backward_Action, Output_Shape_Calculator > >
- struct is_value

- struct is_value< value< T > >
- struct is_variable
- struct is_variable< variable< Tsor > >
- struct model
- struct place_holder
- struct place_holder_state
- struct regularizer
- struct rmsprop
- struct sgd
- struct tensor
- struct tensor_deduction
- struct unary_operator

  A unary operator is composed of a.) an input expression, b.) a forward action and c.) a backward action.
- struct value
- struct variable
- struct variable_state

Concepts

- concept Complex

  A type that represents a complex expression.
- concept Constant
- concept Unary_Operator

  A type that represents an unary operator.
- concept Binary_Operator

  A type that represents a binary operator.
- concept Operator

  A type that represents an unary or a binary operator.
- concept Expression

  A type that represents a unary operator, a binary operator, a variable, a place_holder, a constant or a value.
- concept Place_Holder
- concept Tensor
- concept Value
- concept Variable

Typedefs

- template<typename Loss , typename T >
  using ada_grad = adagrad< Loss, T >
- template<typename Loss , typename T >
  using rms_prop = rmsprop< Loss, T >
- template<typename Loss , typename T >
  using ada_delta = adadelta< Loss, T >
- template<typename T >
  using default_allocator = buffered_allocator< T, 256 >

Functions

- template<std::floating_point Float>
  constexpr auto heaviside_step (Float f) noexcept

  Step activation function, an unary operator.
- template<Expression Ex>
  constexpr auto soft_sign (Ex const &ex) noexcept
- template<Expression Ex>
  constexpr auto unit_step (Ex const &ex) noexcept
- template<Expression Ex>
  constexpr auto binary_step (Ex const &ex) noexcept
- template<Expression Ex>
  constexpr auto gaussian (Ex const &ex) noexcept

  Gaussian activation function, an unary operator.
- template<Expression Ex>
  constexpr auto softmax (Ex const &ex) noexcept

  Softmax activation function, an unary operator.
- template<Expression Ex>
  auto selu (Ex const &ex) noexcept

  Scaled Exponential Linear Unit (SELU) activation function, an unary operator. If `x>0`*, returns 1.0507 x; Otherwise, returns 1.67326*1.0507*(exp(x)-1)*
- template<Expression Ex>
  auto softplus (Ex const &ex) noexcept

  Softplus function, an unary operator. Returns `log(exp(x)+1).`
- template<Expression Ex>
  auto softsign (Ex const &ex) noexcept

  Softsign function, an unary operator. Returns `x / (abs(x) + 1).`
- template<Expression Ex>
  auto sigmoid (Ex const &ex) noexcept

  Sigmoid function, an unary operator. Returns `1 / (exp(-x) + 1).`
- template<Expression Ex>
  auto relu (Ex const &ex) noexcept

  Relu function, an unary operator. Returns `x` *if positive,* `0` *otherwise.*
- template<Expression Ex>
  auto relu6 (Ex const &ex) noexcept

  Rectified Linear 6 function, an unary operator. Returns `min(max(features, 0), 6).`
- template<typename T >
  requires std::floating_point<T>
  auto leaky_relu (T const factor=0.2) noexcept

  Leaky Rectified Linear function, an unary operator. Returns `x` *if positive,* `alpha x` *otherwise.* `alpha` *defaults to 0.2.*
- template<typename T >
  requires std::floating_point<T>
  auto prelu (T const factor) noexcept
- template<Expression Ex>
  auto negative_relu (Ex const &ex) noexcept
- template<typename T = float>
  requires std::floating_point<T>
  auto elu (T const alpha=1.0) noexcept

  Exponential Linear function, an unary operator. Returns `x` *if positive,* `alpha* (exp(x)-1)` *otherwise.* `alpha` *defaults to 0.2.*
- template<Expression Ex>
  auto exponential (Ex const &ex) noexcept

  Exponential function, an unary operator. Returns `exp(x).`

- template<Expression Ex>
  auto hard_sigmoid (Ex const &ex) noexcept

  Hard Sigmoid function, an unary operator. Piecewise linear approximation of the sigmoid function.
- template<Expression Ex>
  auto gelu (Ex const &ex) noexcept

  Gaussian Error function, an unary operator. GAUSSIAN ERROR LINEAR UNITS (GELUS) `https://arxiv.org/pdf/1606.08415.pdf` *$f(x) = 0.5x (1 + tanh[\sqrt{2/\pi}(x + 0.044715x^3)])$ $df = x (1 + tanh[\sqrt{2/\pi}(x + 0.044715x^3)]) + \sqrt(2/\pi) x sech^2[\sqrt(2/\pi) x (1+0.44715x^2) (1+0.134145x^2)]$ where $sec^2(x) = 1 - tanh^2(x)$ derivative generated using service from* `https://www.symbolab.com/solver/derivative-calculator`.
- template<Expression Ex>
  auto swish (Ex const &ex) noexcept

  Swish activation function.
- template<Expression Ex>
  auto silu (Ex const &ex) noexcept

  An alias name of activation swish.
- template<Expression Ex>
  auto crelu (Ex const &ex) noexcept

  Concatenated Rectified Linear Units, an activation function which preserves both positive and negative phase information while enforcing non-saturated non-linearity.
- template<Expression Ex>
  auto tank_shrink (Ex const &ex) noexcept

  Tank shrink function.
- template<Expression Ex>
  auto mish (Ex const &ex) noexcept

  Mish function.
- template<Expression Ex>
  auto lisht (Ex const &ex) noexcept

  Lisht function.
- template<Expression Real_Ex, Expression Imag_Ex>
  Real_Ex real (complex< Real_Ex, Imag_Ex > const &c) noexcept
- template<Expression Real_Ex, Expression Imag_Ex>
  Imag_Ex imag (complex< Real_Ex, Imag_Ex > const &c) noexcept
- template<Complex C>
  auto abs (C const &c) noexcept

  Returns the magnitude of the complex expression.
- template<Complex C>
  auto norm (C const &c) noexcept

  Returns the squared magnitude of the complex expression.
- template<Complex C>
  auto conj (C const &c) noexcept

  Returns the conjugate of the complex expression.
- template<Expression Em, Expression Ep>
  auto polar (Em const &em, Ep const &ep) noexcept

  Returns with given magnitude and phase angle.
- template<Complex C>
  auto arg (C const &c) noexcept

  Calculates the phase angle (in radians) of the complex expression.
- template<Complex C>
  auto operator+ (C const &c) noexcept

  Returns the complex expression.
- template<Complex C>
  auto operator- (C const &c) noexcept

  Negatives the complex expression.

- template<Complex Cl, Complex Cr>
  auto operator+ (Cl const &cl, Cr const &cr) noexcept

  Sums up two complex expressions.
- template<Complex Cl, Complex Cr>
  auto operator- (Cl const &cl, Cr const &cr) noexcept

  Subtracts one complex expression from the other one.
- template<Complex Cl, Complex Cr>
  auto operator* (Cl const &cl, Cr const &cr) noexcept

  Multiplies two complex expressions. Optimization here: (a+ib)(c+id) = (ac-bd) + i(ad+bc) = (ac-bd) + i((a+b)*(c+d)-ac-bd)*
- template<Complex C, Expression E>
  auto operator+ (C const &c, E const &e) noexcept

  Sums up a complex expression and an expression.
- template<Complex C, Expression E>
  auto operator+ (E const &e, C const &c) noexcept

  Sums up a complex expression and an expression.
- template<Complex C, Expression E>
  auto operator- (C const &c, E const &e) noexcept

  Subtracts an expression from a compression expression.
- template<Complex C, Expression E>
  auto operator- (E const &e, C const &c) noexcept

  Subtractsa complex expression from an expression.
- template<Complex C, Expression E>
  auto operator* (C const &c, E const &e) noexcept

  Multiplies a complex expression with an expression.
- template<Complex C, Expression E>
  auto operator* (E const &e, C const &c) noexcept

  Multiplies an expression with a compression expression.
- auto Input (std::vector< unsigned long > const &input_shape={{-1UL}})
- auto Conv2D (unsigned long output_channels, std::vector< unsigned long > const &kernel_size, std::vector< unsigned long > const &input_shape, std::string const &padding="valid", std::vector< unsigned long > const &strides={1, 1}, std::vector< unsigned long > const &dilations={1, 1}, bool use_bias=true, float kernel_regularizer_l1=0.0f, float kernel_regularizer_l2=0.0f, float bias_regularizer_l1=0.0f, float bias_regularizer_l2=0.0f) noexcept

  2D convolution layer.
- auto Conv2D (unsigned long output_channels, std::vector< unsigned long > const &kernel_size, std::string const &padding="valid", std::vector< unsigned long > const &strides={1, 1}, std::vector< unsigned long > const &dilations={1, 1}, bool use_bias=true, float kernel_regularizer_l1=0.0f, float kernel_regularizer_l2=0.0f, float bias_regularizer_l1=0.0f, float bias_regularizer_l2=0.0f) noexcept

  2D convolution layer.
- auto Dense (unsigned long output_size, bool use_bias=true, float kernel_regularizer_l1=0.0f, float kernel_regularizer_l2=0.0f, float bias_regularizer_l1=0.0f, float bias_regularizer_l2=0.0f)

  Densly-connected layer.
- auto BatchNormalization (float threshold=0.95f, float kernel_regularizer_l1=0.0f, float kernel_regularizer_l2=0.0f, float bias_regularizer_l1=0.0f, float bias_regularizer_l2=0.0f)

  Applies a transformation that maintains the mean output close to 0 and the output standard deviation close to 1.
- auto Concatenate (unsigned long axis=-1) noexcept
- auto Add () noexcept
- auto Subtract () noexcept
- auto Multiply () noexcept
- template<Expression Ex>
  auto ReLU (Ex const &ex) noexcept
- auto Softmax () noexcept

- template<typename T = float>
  auto LeakyReLU (T const factor=0.2) noexcept
- template<typename T = float>
  auto ELU (T const factor=0.2) noexcept
- auto Reshape (std::vector< unsigned long > const &new_shape, bool include_batch_flag=true) noexcept
- auto Flatten () noexcept
- auto MaxPooling2D (unsigned long stride) noexcept
- auto UpSampling2D (unsigned long stride) noexcept
- template<typename T >
  auto Dropout (T factor) noexcept
- auto AveragePooling2D (unsigned long stride) noexcept
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto mean_squared_logarithmic_error (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto squared_loss (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto mean_squared_error (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto mse (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto abs_loss (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto mean_absolute_error (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto mae (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto cross_entropy (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto binary_cross_entropy_loss (Lhs_Expression const &ground_truth, Rhs_Expression const &prediction) noexcept
- template<Expression Lhs_Expression, Expression Rhs_Expression, std::floating_point F = float>
  constexpr auto cross_entropy_loss (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex, F label_smoothing_factor=0.0) noexcept
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto hinge_loss (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept
- template<Expression Lhs_Expression, Expression Rhs_Expression, std::floating_point FP>
  auto binary_accuracy (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex, FP threshold=0.5) noexcept
- template<Expression Ex>
  void make_trainable (Ex &ex, bool t)
- template<Expression Ex, Place_Holder Ph, Expression Ey>
  auto replace_placeholder_with_expression (Ex const &ex, Ph const &old_place_holder, Ey const &new_expression)
- template<typename Model , typename Optimizer , typename Loss >
  auto make_compiled_model (Model const &m, Loss const &l, Optimizer const &o)
- template<typename Forward_Action , typename Backward_Action , typename Output_Shape_Calculator = identity_output_shape_calculator>
  constexpr auto make_unary_operator (Forward_Action const &unary_forward_action, Backward_Action const &unary_backward_action, std::string const &name="Anonymous Unary Operator", Output_Shape_Calculator const &output_shape_calculator=Output_Shape_Calculator{}) noexcept

  Construct an unary operator by passing the forward/backward actions and output shape calculator.

- template<typename Forward_Action , typename Backward_Action , typename Output_Shape_Calculator = identity_output_shape_calculator>
  auto make_binary_operator (Forward_Action const &binary_forward_action, Backward_Action const &binary_backward_action, std::string const &name="Anonymous Binary Operator", Output_Shape_Calculator const &output_shape_calculator=Output_Shape_Calculator{}) noexcept
- template<Expression Ex>
  std::string computation_graph (Ex const &ex) noexcept
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto plus (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto operator+ (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept
- template<Expression Ex>
  constexpr auto operator+ (Ex const &ex) noexcept
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  auto operator* (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept
- template<Expression Ex>
  constexpr auto negative (Ex const &ex) noexcept

  Negative operator, elementwise.
- template<Expression Ex>
  constexpr auto operator- (Ex const &ex) noexcept
- template<Expression Ex>
  constexpr auto inverse (Ex const &ex) noexcept

  Inverse operator, elementwise.
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto elementwise_product (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept

  Multiply two input operators, elementwise.
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto elementwise_multiply (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto hadamard_product (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto divide (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept

  Divide one tensor by the other.
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto operator/ (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept

  Divide one tensor by the other.
- template<Expression Ex>
  constexpr auto sum_reduce (Ex const &ex) noexcept

  Sum up all elements, returns a scalar.
- template<Expression Ex>
  constexpr auto reduce_sum (Ex const &ex) noexcept
- template<Expression Ex>
  constexpr auto mean_reduce (Ex const &ex) noexcept

  Computes the mean of elements across all dimensions of an expression.
- template<Expression Ex>
  constexpr auto reduce_mean (Ex const &ex) noexcept

  An alias name of mean_reduce.
- template<Expression Ex>
  constexpr auto mean (Ex const &ex) noexcept

  An alias name of mean_reduce.
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto minus (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept

- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto operator- (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept
- template<Expression Ex>
  constexpr auto square (Ex const &ex) noexcept
- template<Place_Holder Ph>
  bool operator== (Ph const &lhs, Ph const &rhs)
- template<Place_Holder Ph>
  bool operator!= (Ph const &lhs, Ph const &rhs)
- template<Place_Holder Ph>
  bool operator< (Ph const &lhs, Ph const &rhs)
- template<Place_Holder Ph>
  bool operator> (Ph const &lhs, Ph const &rhs)
- template<Place_Holder Ph>
  bool operator<= (Ph const &lhs, Ph const &rhs)
- template<Place_Holder Ph>
  bool operator>= (Ph const &lhs, Ph const &rhs)
- auto lstm (std::unsigned long units) noexcept
- template<Tensor Tsor>
  ceras_private::session< Tsor > & get_default_session ()

  Get the default global session.
- template<Tensor Tsor>
  auto & bind (place_holder< Tsor > &p_holder, Tsor const &value)

  Bind a tensor to a place holder.
- template<typename Operation >
  auto run (Operation &op)

  Run an expression.
- template<typename T , typename A = default_allocator<T>>
  constexpr tensor< T, A > as_tensor (T val) noexcept
- template<Variable Var>
  bool operator== (Var const &lhs, Var const &rhs) noexcept

Variables

- template<typename T >
  constexpr bool is_complex_v = is_complex<T>::value
- template<class T >
  constexpr bool is_constant_v = is_constant<T>::value
- static constexpr auto MeanSquaredError

  Computes the mean of squares of errors between labels and predictions.
- static constexpr auto MSE

  An alias name of function MeanSquaredError.
- static constexpr auto MeanAbsoluteError

  Computes the mean of absolute errors between labels and predictions.
- static constexpr auto MAE

  An alias name of function MeanAbsoluteError.
- static constexpr auto Hinge
- static constexpr auto CategoricalCrossentropy
- static constexpr auto CategoricalCrossEntropy
- static constexpr auto BinaryCrossentropy
- static constexpr auto BinaryCrossEntropy
- template<class T >
  constexpr bool is_unary_operator_v = is_unary_operator<T>::value
- template<class T >
  constexpr bool is_binary_operator_v = is_binary_operator<T>::value

- auto Adam
- auto SGD
- auto Adagrad
- auto RMSprop
- auto Adadelta
- template<class T >
  constexpr bool is_place_holder_v = is_place_holder<T>::value
- static unsigned long random_seed = std::chrono::system_clock::now().time_since_epoch().count()

  Random seed for the tensor library.
- static std::mt19937 random_generator {random_seed}
- template<class T >
  constexpr bool is_tensor_v = is_tensor<T>::value
- template<class T >
  constexpr bool is_value_v = is_value<T>::value
- template<class T >
  constexpr bool is_variable_v = is_variable<T>::value

6.1.1 Typedef Documentation

6.1.1.1 ada_delta

```
template<typename Loss , typename T >
using ceras::ada_delta = typedef adadelta< Loss, T >
```

6.1.1.2 ada_grad

```
template<typename Loss , typename T >
using ceras::ada_grad = typedef adagrad<Loss, T>
```

6.1.1.3 default_allocator

```
template<typename T >
using ceras::default_allocator = typedef buffered_allocator<T, 256>
```

6.1.1.4 rms_prop

```
template<typename Loss , typename T >
using ceras::rms_prop = typedef rmsprop< Loss, T >
```

6.1.2 Function Documentation

6.1.2.1 abs()

```
template<Complex C>
auto ceras::abs (
            C const & c )  [noexcept]
```

Returns the magnitude of the complex expression.

Parameters

c	Complex expression.

```
auto r = variable{ ...  };
auto i = variable{ ...  };
auto c = complex{ r, i };
auto a = abs( c );
```

6.1.2.2 abs_loss()

```
template<Expression Lhs_Expression, Expression Rhs_Expression>
constexpr auto ceras::abs_loss (
            Lhs_Expression const & lhs_ex,
            Rhs_Expression const & rhs_ex )  [constexpr], [noexcept]
```

6.1.2.3 Add()

```
auto ceras::Add ( )  [inline], [noexcept]
```

Layer that adds two layers

Example usage:

```
auto input = Input(); // (16, )
auto x1 = Dense( 8, 16 )( input );
auto x2 = Dense( 8, 16 )( input );
auto x3 = Add()( x1, x2 ); // equivalent to 'x1 + x2'
auto m = model{ input, x3 };
```

6.1.2.4 arg()

```
template<Complex C>
auto ceras::arg (
            C const & c )  [noexcept]
```

Calculates the phase angle (in radians) of the complex expression.

Parameters

c	Complex expression. Implemented as `atan2( imagec), real(c) ).`

```
auto r = variable{ ... };
auto i = variable{ ... };
auto c = complex{ r, i };
auto a = arg( c );
```

6.1.2.5 as_tensor()

```
template<typename T , typename A = default_allocator<T>>
constexpr tensor< T, A > ceras::as_tensor (
            T val ) [constexpr], [noexcept]
```

6.1.2.6 AveragePooling2D()

```
auto ceras::AveragePooling2D (
            unsigned long stride ) [inline], [noexcept]
```

Average pooling operation for spatial data.

6.1.2.7 BatchNormalization()

```
auto ceras::BatchNormalization (
            float threshold = 0.95f,
            float kernel_regularizer_l1 = 0.0f,
            float kernel_regularizer_l2 = 0.0f,
            float bias_regularizer_l1 = 0.0f,
            float bias_regularizer_l2 = 0.0f ) [inline]
```

Applies a transformation that maintains the mean output close to 0 and the output standard deviation close to 1.

Parameters

shape	Dimensionality of the input shape.
threshold	Momentum for the moving average.
kernel_regularizer↵_l1	L1 regularizer for the kernel. Defaults to `0.0f`.
kernel_regularizer↵_l2	L2 regularizer for the kernel. Defaults to `0.0f`.
bias_regularizer_l1	L1 regularizer for the bias vector. Defaults to `0.0f`.
bias_regularizer_l2	L2 regularizer for the bias vector. Defaults to `0.0f`.

Example code:

```
auto a = variable{ random<float>( {12, 34, 56, 78} ) };
auto b = BatchNormalization( {34, 56, 78}, 0.8f )( a ); // note the leading dimension of 'a' is intepretated
      as batch size, and only the rest 3 dimensions are required here.
```

Applies a transformation that maintains the mean output close to 0 and the output standard deviation close to 1.

Parameters

threshold	Momentum for the moving average.
kernel_regularizer_l1	L1 regularizer for the kernel. Defaults to `0.0f`.
kernel_regularizer_l2	L2 regularizer for the kernel. Defaults to `0.0f`.
bias_regularizer_l1	L1 regularizer for the bias vector. Defaults to `0.0f`.
bias_regularizer_l2	L2 regularizer for the bias vector. Defaults to `0.0f`.

Example code:

```
auto a = variable{ random<float>( {12, 34, 56, 78} ) };
auto b = BatchNormalization( {34, 56, 78}, 0.8f )( a ); // note the leading dimension of 'a' is intepretated
      as batch size, and only the rest 3 dimensions are required here.
```

6.1.2.8 binary_accuracy()

```
template<Expression Lhs_Expression, Expression Rhs_Expression, std::floating_point FP>
auto ceras::binary_accuracy (
            Lhs_Expression const & lhs_ex,
            Rhs_Expression const & rhs_ex,
            FP threshold = 0.5 )  [noexcept]
```

6.1.2.9 binary_cross_entropy_loss()

```
template<Expression Lhs_Expression, Expression Rhs_Expression>
constexpr auto ceras::binary_cross_entropy_loss (
            Lhs_Expression const & ground_truth,
            Rhs_Expression const & prediction )  [constexpr], [noexcept]
```

6.1.2.10 binary_step()

```
template<Expression Ex>
constexpr auto ceras::binary_step (
            Ex const & ex )  [constexpr], [noexcept]
```

6.1.2.11 bind()

```
template<Tensor Tsor>
auto & ceras::bind (
            place_holder< Tsor > & p_holder,
            Tsor const & value )
```

Bind a tensor to a place holder.

Parameters

p_holder	The place holder.
value	The tensor to bind.

Returns

A default session.

6.1.2.12 computation_graph()

```
template<Expression Ex>
std::string ceras::computation_graph (
            Ex const & ex ) [inline], [noexcept]
```

Generating the computation graph, in `graph description language`.

Parameters

ex	An expression.

Returns

A string describing the computation graph, in graph description language.

6.1.2.13 Concatenate()

```
auto ceras::Concatenate (
            unsigned long axis = -1 ) [inline], [noexcept]
```

Layer that concatenates two layers.

Parameters

axis	The concatenation axis. Default to the last channel.

Example usage:

```
auto l1 = variable{ tensor<float>{ {12, 11, 3} } };
auto l2 = variable{ tensor<float>{ {12, 11, 4} } };
auto l12 = Concatenate()( l1, l2 ); // should be of shape (12, 11, 7)
```

6.1.2.14 conj()

```
template<Complex C>
```

```
auto ceras::conj (
            C const & c ) [noexcept]
```

Returns the conjugate of the complex expression.

Parameters

c	Complex expression.

```
auto r = variable{ ... };
auto i = variable{ ... };
auto c = complex{ r, i };
auto a = conj( c );
```

6.1.2.15 Conv2D() [1/2]

```
auto ceras::Conv2D (
            unsigned long output_channels,
            std::vector< unsigned long > const & kernel_size,
            std::string const & padding = "valid",
            std::vector< unsigned long > const & strides = {1,1},
            std::vector< unsigned long > const & dilations = {1, 1},
            bool use_bias = true,
            float kernel_regularizer_l1 = 0.0f,
            float kernel_regularizer_l2 = 0.0f,
            float bias_regularizer_l1 = 0.0f,
            float bias_regularizer_l2 = 0.0f ) [inline], [noexcept]
```

2D convolution layer.

Parameters

output_channels	Dimensionality of the output space.
kernel_size	The height and width of the convolutional window.
padding	`valid` or `same`. `valid` suggests no padding. `same` suggests zero padding. Defaults to `valid`.
strides	The strides along the height and width direction. Defaults to `(1, 1)`.
dilations	The dialation along the height and width direction. Defaults to `(1, 1)`.
use_bias	Wether or not use a bias vector. Defaults to `true`.
kernel_regularizer←_l1	L1 regularizer for the kernel. Defaults to `0.0f`.
kernel_regularizer←_l2	L2 regularizer for the kernel. Defaults to `0.0f`.
bias_regularizer_l1	L1 regularizer for the bias vector. Defaults to `0.0f`.
bias_regularizer_l2	L2 regularizer for the bias vector. Defaults to `0.0f`.

Example code:

```
auto x = Input{ {28, 28, 1} };
auto y = Conv2D( 32, {3, 3}, "same" )( x );
auto z = Flatten()( y );
auto u = Dense( 10, 28*28*32 )( z );
auto m = model{ x, u };
```

6.1.2.16 Conv2D() [2/2]

```
auto ceras::Conv2D (
            unsigned long output_channels,
            std::vector< unsigned long > const & kernel_size,
            std::vector< unsigned long > const & input_shape,
            std::string const & padding = "valid",
            std::vector< unsigned long > const & strides = {1,1},
            std::vector< unsigned long > const & dilations = {1, 1},
            bool use_bias = true,
            float kernel_regularizer_l1 = 0.0f,
            float kernel_regularizer_l2 = 0.0f,
            float bias_regularizer_l1 = 0.0f,
            float bias_regularizer_l2 = 0.0f )  [inline], [noexcept]
```

2D convolution layer.

Parameters

output_channels	Dimensionality of the output space.
kernel_size	The height and width of the convolutional window.
input_shape	Dimensionality of the input shape.
padding	`valid` or `same`. `valid` suggests no padding. `same` suggests zero padding. Defaults to `valid`.
strides	The strides along the height and width direction. Defaults to `(1, 1)`.
dilations	The dialation along the height and width direction. Defaults to `(1, 1)`.
use_bias	Wether or not use a bias vector. Defaults to `true`.
kernel_regularizer↩ _l1	L1 regularizer for the kernel. Defaults to `0.0f`.
kernel_regularizer↩ _l2	L2 regularizer for the kernel. Defaults to `0.0f`.
bias_regularizer_l1	L1 regularizer for the bias vector. Defaults to `0.0f`.
bias_regularizer_l2	L2 regularizer for the bias vector. Defaults to `0.0f`.

Example code:

```
auto x = Input{};
auto y = Conv2D( 32, {3, 3}, {28, 28, 1}, "same" )( x );
auto z = Flatten()( y );
auto u = Dense( 10, 28*28*32 )( z );
auto m = model{ x, u };
```

6.1.2.17 crelu()

```
template<Expression Ex>
auto ceras::crelu (
            Ex const & ex )  [noexcept]
```

Concatenated Rectified Linear Units, an activation function which preserves both positive and negative phase information while enforcing non-saturated non-linearity.

Reference: Shang, Wenling, Kihyuk Sohn, Diogo Almeida, and Honglak Lee. “Understanding and Improving Convolutional Neural Networks via Concatenated Rectified Linear Units.” ArXiv:1603.05201 [Cs], July 19, 2016. http://arxiv.org/abs/1603.05201.

```
auto v = variable{ random<float>{ 3, 3 } };
auto c = crelu( v );
```

6.1.2.18 cross_entropy()

```
template<Expression Lhs_Expression, Expression Rhs_Expression>
constexpr auto ceras::cross_entropy (
            Lhs_Expression const & lhs_ex,
            Rhs_Expression const & rhs_ex )  [constexpr], [noexcept]
```

6.1.2.19 cross_entropy_loss()

```
template<Expression Lhs_Expression, Expression Rhs_Expression, std::floating_point F = float>
constexpr auto ceras::cross_entropy_loss (
            Lhs_Expression const & lhs_ex,
            Rhs_Expression const & rhs_ex,
            F label_smoothing_factor = 0.0 )  [constexpr], [noexcept]
```

6.1.2.20 Dense()

```
auto ceras::Dense (
            unsigned long output_size,
            bool use_bias = true,
            float kernel_regularizer_l1 = 0.0f,
            float kernel_regularizer_l2 = 0.0f,
            float bias_regularizer_l1 = 0.0f,
            float bias_regularizer_l2 = 0.0f )  [inline]
```

Densly-connected layer.

Parameters

output_size	Dimensionality of output shape. The output shape is `(batch_size, output_size)`.
input_size	Dimensionality of input shape. The input shape is `(batch_size, input_size)`.
use_bias	Using a bias vector or not. Defaults to `true`.
kernel_regularizer↩_l1	L1 regularizer for the kernel. Defaults to `0.0f`.
kernel_regularizer↩_l2	L2 regularizer for the kernel. Defaults to `0.0f`.
bias_regularizer_l1	L1 regularizer for the bias vector. Defaults to `0.0f`.
bias_regularizer_l2	L2 regularizer for the bias vector. Defaults to `0.0f`.

Example code:

```
auto x = Input{ {28*28,} };
auto y = Dense( 10, 28*28 )( x );
auto m = model{ x, y };
```

Densly-connected layer.

Parameters

output_size	Dimensionality of output shape. The output shape is (batch_size, output_size).
use_bias	Using a bias vector or not. Defaults to true.
kernel_regularizer←↩ _l1	L1 regularizer for the kernel. Defaults to 0.0f.
kernel_regularizer←↩ _l2	L2 regularizer for the kernel. Defaults to 0.0f.
bias_regularizer_l1	L1 regularizer for the bias vector. Defaults to 0.0f.
bias_regularizer_l2	L2 regularizer for the bias vector. Defaults to 0.0f.

Example code:
```
auto x = Input{ {28*28,} };
auto y = Dense( 10, )( x );
auto m = model{ x, y };
```

6.1.2.21 divide()

```
template<Expression Lhs_Expression, Expression Rhs_Expression>
constexpr auto ceras::divide (
            Lhs_Expression const & lhs_ex,
            Rhs_Expression const & rhs_ex )  [constexpr], [noexcept]
```

Divide one tensor by the other.
```
auto x = varialbe{ tensor<float>{ {17, 12} } };
auto y = varialbe{ tensor<float>{ {17, 12} } };
auto z = divide( x, y ); // z = x / y
```

6.1.2.22 Dropout()

```
template<typename T >
auto ceras::Dropout (
            T factor )  [inline], [noexcept]
```

Applies Dropout to the input.

6.1.2.23 elementwise_multiply()

```
template<Expression Lhs_Expression, Expression Rhs_Expression>
constexpr auto ceras::elementwise_multiply (
            Lhs_Expression const & lhs_ex,
            Rhs_Expression const & rhs_ex )  [constexpr], [noexcept]
```

6.1.2.24 elementwise_product()

```
template<Expression Lhs_Expression, Expression Rhs_Expression>
constexpr auto ceras::elementwise_product (
            Lhs_Expression const & lhs_ex,
            Rhs_Expression const & rhs_ex ) [constexpr], [noexcept]
```

Multiply two input operators, elementwise.

```
auto x = variable{ tensor<float>{ {2, 3, 5} } };
auto y = variable{ tensor<float>{ {2, 3, 5} } };
auto z = elementwise_product( x, y ); // z = x*y;
```

6.1.2.25 elu()

```
template<typename T = float>
requires std::floating_point<T>
auto ceras::elu (
            T const alpha = 1.0 ) [noexcept]
```

Exponential Linear function, an unary operator. Returns `x` if positive, `alpha* (exp(x)-1)` otherwise. `alpha` defaults to 0.2.

Parameters

ex	An input operator.

```
auto x = Input();
auto y = Dense( 10, 28*28 )( x );
auto output = elu(0.1f)( y );
```

6.1.2.26 ELU()

```
template<typename T = float>
auto ceras::ELU (
            T const factor = 0.2 ) [inline], [noexcept]
```

Exponential Linear Unit.

6.1.2.27 exponential()

```
template<Expression Ex>
auto ceras::exponential (
            Ex const & ex ) [inline], [noexcept]
```

Exponential function, an unary operator. Returns `exp(x)`.

Parameters

ex	An input operator.

```
auto x = Input();
auto y = Dense( 10, 28*28 )( x );
auto output = exponential( y );
```

6.1.2.28 Flatten()

```
auto ceras::Flatten ( ) [inline], [noexcept]
```

Flattens the input. Does not affect the batch size.

6.1.2.29 gaussian()

```
template<Expression Ex>
constexpr auto ceras::gaussian (
            Ex const & ex ) [constexpr], [noexcept]
```

Gaussian activation function, an unary operator.

Parameters

ex	An input operator

```
auto x = Input();
auto y = Dense( 10, 28*28 )( x );
auto output = gaussian( y );
```

6.1.2.30 gelu()

```
template<Expression Ex>
auto ceras::gelu (
            Ex const & ex ) [inline], [noexcept]
```

Gaussian Error function, an unary operator. GAUSSIAN ERROR LINEAR UNITS (GELUS) https://arxiv.org/pdf/1606.08415.pdf $f(x) = 0.5x (1 + tanh[\sqrt{2/\pi}(x + 0.044715x^3)])$ $df = x (1 + tanh[\sqrt{2/\pi}(x + 0.044715x^3)]) + \sqrt(2/\pi) x sech^2[\sqrt(2/\pi) x (1+0.44715x^2) (1+0.134145x^2)]$ where $sec^2(x) = 1 - tanh^2(x)$ derivative generated using service from https://www.symbolab.com/solver/derivative-calculator.

Parameters

ex	An input operator.

```
auto x = Input();
auto y = Dense( 10, 28*28 )( x );
auto output = gelu( y );
```

6.1.2.31 get_default_session()

```
template<Tensor Tsor>
ceras_private::session< Tsor > & ceras::get_default_session ( )
```

Get the default global session.

6.1.2.32 hadamard_product()

```
template<Expression Lhs_Expression, Expression Rhs_Expression>
constexpr auto ceras::hadamard_product (
            Lhs_Expression const & lhs_ex,
            Rhs_Expression const & rhs_ex )  [constexpr], [noexcept]
```

6.1.2.33 hard_sigmoid()

```
template<Expression Ex>
auto ceras::hard_sigmoid (
            Ex const & ex )  [inline], [noexcept]
```

Hard Sigmoid function, an unary operator. Piecewise linear approximation of the sigmoid function.

Parameters

ex	An input operator.

```
auto x = Input();
auto y = Dense( 10, 28*28 )( x );
auto output = hard_sigmoid( y );
```

6.1.2.34 heaviside_step()

```
template<std::floating_point Float>
constexpr auto ceras::heaviside_step (
            Float f )  [constexpr], [noexcept]
```

Step activation function, an unary operator.

Parameters

ex	An input operator

```
auto x = Input();
auto y = Dense( 10, 28*28 )( x );
auto output = heaviside_step( y );
```

6.1.2.35 hinge_loss()

```
template<Expression Lhs_Expression, Expression Rhs_Expression>
constexpr auto ceras::hinge_loss (
            Lhs_Expression const & lhs_ex,
            Rhs_Expression const & rhs_ex )  [constexpr], [noexcept]
```

6.1.2.36 imag()

```
template<Expression Real_Ex, Expression Imag_Ex>
Imag_Ex ceras::imag (
            complex< Real_Ex, Imag_Ex > const & c )  [noexcept]
```

@bref Returns the imaginary part of the complex expression.

Parameters

c	A complex expression.

6.1.2.37 Input()

```
auto ceras::Input (
            std::vector< unsigned long > const & input_shape = {{-1UL}} )  [inline]
```

6.1.2.38 inverse()

```
template<Expression Ex>
constexpr auto ceras::inverse (
            Ex const & ex )  [constexpr], [noexcept]
```

Inverse operator, elementwise.

```
auto x = variable{ ... };
auto ix = inverse( x );
```

6.1.2.39 leaky_relu()

```
template<typename T >
requires std::floating_point<T>
auto ceras::leaky_relu (
            T const factor = 0.2 )  [noexcept]
```

Leaky Rectified Linear function, an unary operator. Returns `x` if positive, `alpha x` otherwise. `alpha` defaults to 0.2.

Parameters

ex	An input operator.

```
auto x = Input();
auto y = Dense( 10, 28*28 )( x );
auto output = leaky_relu(0.1f)( y );
```

6.1.2.40 LeakyReLU()

```
template<typename T = float>
auto ceras::LeakyReLU (
            T const factor = 0.2 ) [inline], [noexcept]
```

leaky relu activation function.

6.1.2.41 lisht()

```
template<Expression Ex>
auto ceras::lisht (
            Ex const & ex ) [noexcept]
```

Lisht function.

```
auto v = variable{ random<float>{ 3, 3 } };
auto c = lisht( v );
```

6.1.2.42 lstm()

```
auto ceras::lstm (
            std::unsigned long units ) [inline], [noexcept]
```

6.1.2.43 mae()

```
template<Expression Lhs_Expression, Expression Rhs_Expression>
constexpr auto ceras::mae (
            Lhs_Expression const & lhs_ex,
            Rhs_Expression const & rhs_ex ) [constexpr], [noexcept]
```

6.1.2.44 make_binary_operator()

```
template<typename Forward_Action , typename Backward_Action , typename Output_Shape_Calculator
= identity_output_shape_calculator>
auto ceras::make_binary_operator (
            Forward_Action const & binary_forward_action,
            Backward_Action const & binary_backward_action,
            std::string const & name = "Anonymous Binary Operator",
            Output_Shape_Calculator const & output_shape_calculator = Output_Shape_Calculator{}
)  [noexcept]
```

6.1.2.45 make_compiled_model()

```
template<typename Model , typename Optimizer , typename Loss >
auto ceras::make_compiled_model (
            Model const & m,
            Loss const & l,
            Optimizer const & o )  [inline]
```

6.1.2.46 make_trainable()

```
template<Expression Ex>
void ceras::make_trainable (
            Ex & ex,
            bool t )
```

Setting an expression's trainable flag

6.1.2.47 make_unary_operator()

```
template<typename Forward_Action , typename Backward_Action , typename Output_Shape_Calculator
= identity_output_shape_calculator>
constexpr auto ceras::make_unary_operator (
            Forward_Action const & unary_forward_action,
            Backward_Action const & unary_backward_action,
            std::string const & name = "Anonymous Unary Operator",
            Output_Shape_Calculator const & output_shape_calculator = Output_Shape_Calculator{}
)  [constexpr], [noexcept]
```

Construct an unary operator by passing the forward/backward actions and output shape calculator.

6.1.2.48 MaxPooling2D()

```
auto ceras::MaxPooling2D (
            unsigned long stride )  [inline], [noexcept]
```

Max pooling operation for 2D spatial data.

6.1.2.49 mean()

```
template<Expression Ex>
constexpr auto ceras::mean (
            Ex const & ex ) [constexpr], [noexcept]
```

An alias name of mean_reduce.

6.1.2.50 mean_absolute_error()

```
template<Expression Lhs_Expression, Expression Rhs_Expression>
constexpr auto ceras::mean_absolute_error (
            Lhs_Expression const & lhs_ex,
            Rhs_Expression const & rhs_ex ) [constexpr], [noexcept]
```

6.1.2.51 mean_reduce()

```
template<Expression Ex>
constexpr auto ceras::mean_reduce (
            Ex const & ex ) [constexpr], [noexcept]
```

Computes the mean of elements across all dimensions of an expression.

Parameters

ex	Incoming expression.

Example code:

```
auto va = place_holder<tensor<float>>{};
auto vb = variable{ random<float>{ 3, 4} };
auto diff = mean_reduce( va, vb );
```

6.1.2.52 mean_squared_error()

```
template<Expression Lhs_Expression, Expression Rhs_Expression>
constexpr auto ceras::mean_squared_error (
            Lhs_Expression const & lhs_ex,
            Rhs_Expression const & rhs_ex ) [constexpr], [noexcept]
```

6.1.2.53 mean_squared_logarithmic_error()

```
template<Expression Lhs_Expression, Expression Rhs_Expression>
constexpr auto ceras::mean_squared_logarithmic_error (
            Lhs_Expression const & lhs_ex,
            Rhs_Expression const & rhs_ex ) [constexpr], [noexcept]
```

6.1.2.54 minus()

```
template<Expression Lhs_Expression, Expression Rhs_Expression>
constexpr auto ceras::minus (
            Lhs_Expression const & lhs_ex,
            Rhs_Expression const & rhs_ex )  [constexpr], [noexcept]
```

6.1.2.55 mish()

```
template<Expression Ex>
auto ceras::mish (
            Ex const & ex )  [noexcept]
```

Mish function.

```
auto v = variable{ random<float>{ 3, 3 } };
auto c = mish( v );
```

6.1.2.56 mse()

```
template<Expression Lhs_Expression, Expression Rhs_Expression>
constexpr auto ceras::mse (
            Lhs_Expression const & lhs_ex,
            Rhs_Expression const & rhs_ex )  [constexpr], [noexcept]
```

6.1.2.57 Multiply()

```
auto ceras::Multiply ( )  [inline], [noexcept]
```

Layer that elementwise multiplies two layers

Example usage:

```
auto input = Input(); // (16, )
auto x1 = Dense( 8, 16 )( input );
auto x2 = Dense( 8, 16 )( input );
auto x3 = Multiply()( x1, x2 ); // equivalent to `elementwise_multiply(x1, x2)`
auto m = model{ input, x3 };
```

6.1.2.58 negative()

```
template<Expression Ex>
constexpr auto ceras::negative (
            Ex const & ex )  [constexpr], [noexcept]
```

Negative operator, elementwise.

```
auto x = variable{ ...  };
auto ix = negative( x );
```

6.1.2.59 negative_relu()

```
template<Expression Ex>
auto ceras::negative_relu (
            Ex const & ex )  [noexcept]
```

6.1.2.60 norm()

```
template<Complex C>
auto ceras::norm (
            C const & c )  [noexcept]
```

Returns the squared magnitude of the complex expression.

Parameters

c	Complex expression.

```
auto r = variable{ ...  };
auto i = variable{ ...  };
auto c = complex{ r, i };
auto a = norm( c );
```

6.1.2.61 operator"!=()

```
template<Place_Holder Ph>
bool ceras::operator!= (
            Ph const & lhs,
            Ph const & rhs )
```

6.1.2.62 operator*() [1/4]

```
template<Complex C, Expression E>
auto ceras::operator* (
            C const & c,
            E const & e )  [noexcept]
```

Multiplies a complex expression with an expression.

6.1.2.63 operator*() [2/4]

```
template<Complex Cl, Complex Cr>
auto ceras::operator* (
            Cl const & cl,
            Cr const & cr ) [noexcept]
```

Multiplies two complex expressions. Optimization here: (a+ib)*(c+id) = (ac-bd) + i(ad+bc) = (ac-bd) + i((a+b)*(c+d)-ac-bd)

```
auto c1 = complex{ ..., ... };
auto c2 = complex{ ..., ... };
auto c12 = c1 * c2;
```

6.1.2.64 operator*() [3/4]

```
template<Complex C, Expression E>
auto ceras::operator* (
            E const & e,
            C const & c ) [noexcept]
```

Multiplies an expression with a compression expression.

6.1.2.65 operator*() [4/4]

```
template<Expression Lhs_Expression, Expression Rhs_Expression>
auto ceras::operator* (
            Lhs_Expression const & lhs_ex,
            Rhs_Expression const & rhs_ex ) [noexcept]
```

6.1.2.66 operator+() [1/6]

```
template<Complex C>
auto ceras::operator+ (
            C const & c ) [noexcept]
```

Returns the complex expression.

6.1.2.67 operator+() [2/6]

```
template<Complex C, Expression E>
auto ceras::operator+ (
            C const & c,
            E const & e ) [noexcept]
```

Sums up a complex expression and an expression.

6.1.2.68 operator+() [3/6]

```
template<Complex Cl, Complex Cr>
auto ceras::operator+ (
            Cl const & cl,
            Cr const & cr )  [noexcept]
```

Sums up two complex expressions.

6.1.2.69 operator+() [4/6]

```
template<Complex C, Expression E>
auto ceras::operator+ (
            E const & e,
            C const & c )  [noexcept]
```

Sums up a complex expression and an expression.

6.1.2.70 operator+() [5/6]

```
template<Expression Ex>
constexpr auto ceras::operator+ (
            Ex const & ex )  [constexpr], [noexcept]
```

6.1.2.71 operator+() [6/6]

```
template<Expression Lhs_Expression, Expression Rhs_Expression>
constexpr auto ceras::operator+ (
            Lhs_Expression const & lhs_ex,
            Rhs_Expression const & rhs_ex )  [constexpr], [noexcept]
```

6.1.2.72 operator-() [1/6]

```
template<Complex C>
auto ceras::operator- (
            C const & c )  [noexcept]
```

Negatives the complex expression.

6.1.2.73 operator-() [2/6]

```
template<Complex C, Expression E>
auto ceras::operator- (
            C const & c,
            E const & e )  [noexcept]
```

Subtracts an expression from a compression expression.

6.1.2.74 operator-() [3/6]

```
template<Complex Cl, Complex Cr>
auto ceras::operator- (
            Cl const & cl,
            Cr const & cr )  [noexcept]
```

Subtracts one complex expression from the other one.

6.1.2.75 operator-() [4/6]

```
template<Complex C, Expression E>
auto ceras::operator- (
            E const & e,
            C const & c )  [noexcept]
```

Subtractsa complex expression from an expression.

6.1.2.76 operator-() [5/6]

```
template<Expression Ex>
constexpr auto ceras::operator- (
            Ex const & ex )  [constexpr], [noexcept]
```

6.1.2.77 operator-() [6/6]

```
template<Expression Lhs_Expression, Expression Rhs_Expression>
constexpr auto ceras::operator- (
            Lhs_Expression const & lhs_ex,
            Rhs_Expression const & rhs_ex )  [constexpr], [noexcept]
```

6.1.2.78 operator/()

```
template<Expression Lhs_Expression, Expression Rhs_Expression>
constexpr auto ceras::operator/ (
            Lhs_Expression const & lhs_ex,
            Rhs_Expression const & rhs_ex )  [constexpr], [noexcept]
```

Divide one tensor by the other.

```
auto x = varialbe{ tensor<float>{ {17, 12} } };
auto y = varialbe{ tensor<float>{ {17, 12} } };
auto z = x/y; // same as  'divide( x, y );'
```

6.1.2.79 operator<()

```
template<Place_Holder Ph>
bool ceras::operator< (
            Ph const & lhs,
            Ph const & rhs )
```

6.1.2.80 operator<=()

```
template<Place_Holder Ph>
bool ceras::operator<= (
            Ph const & lhs,
            Ph const & rhs )
```

6.1.2.81 operator==() [1/2]

```
template<Place_Holder Ph>
bool ceras::operator== (
            Ph const & lhs,
            Ph const & rhs )
```

6.1.2.82 operator==() [2/2]

```
template<Variable Var>
bool ceras::operator== (
            Var const & lhs,
            Var const & rhs )  [noexcept]
```

6.1.2.83 operator>()

```
template<Place_Holder Ph>
bool ceras::operator> (
            Ph const & lhs,
            Ph const & rhs )
```

6.1.2.84 operator>=()

```
template<Place_Holder Ph>
bool ceras::operator>= (
            Ph const & lhs,
            Ph const & rhs )
```

6.1.2.85 plus()

```
template<Expression Lhs_Expression, Expression Rhs_Expression>
constexpr auto ceras::plus (
            Lhs_Expression const & lhs_ex,
            Rhs_Expression const & rhs_ex )  [constexpr], [noexcept]
```

6.1.2.86 polar()

```
template<Expression Em, Expression Ep>
auto ceras::polar (
            Em const & em,
            Ep const & ep )  [noexcept]
```

Returns with given magnitude and phase angle.

Parameters

em	Magnitude.
ep	Phase.

```
auto r = variable{ ...  };
auto i = variable{ ...  };
auto a = polar( r, i );
```

6.1.2.87 prelu()

```
template<typename T >
requires std::floating_point<T>
```

```
auto ceras::prelu (
            T const factor ) [noexcept]
```

@PReLU is an alias name of Leaky_ReLU

6.1.2.88 real()

```
template<Expression Real_Ex, Expression Imag_Ex>
Real_Ex ceras::real (
            complex< Real_Ex, Imag_Ex > const & c ) [noexcept]
```

@bref Returns the real part of the complex expression.

Parameters

c	A complex expression.

6.1.2.89 reduce_mean()

```
template<Expression Ex>
constexpr auto ceras::reduce_mean (
            Ex const & ex ) [constexpr], [noexcept]
```

An alias name of mean_reduce.

6.1.2.90 reduce_sum()

```
template<Expression Ex>
constexpr auto ceras::reduce_sum (
            Ex const & ex ) [constexpr], [noexcept]
```

6.1.2.91 relu()

```
template<Expression Ex>
auto ceras::relu (
            Ex const & ex ) [noexcept]
```

Relu function, an unary operator. Returns `x` if positive, `0` otherwise.

Parameters

ex	An input operator.

```
auto x = Input();
auto y = Dense( 10, 28*28 )( x );
auto output = relu( y );
```

6.1.2.92 ReLU()

```
template<Expression Ex>
auto ceras::ReLU (
            Ex const & ex )  [inline], [noexcept]
```

Rectified Linear Unit activation function.

6.1.2.93 relu6()

```
template<Expression Ex>
auto ceras::relu6 (
            Ex const & ex )  [noexcept]
```

Rectified Linear 6 function, an unary operator. Returns `min(max(features, 0), 6)`.

Parameters

ex	An input operator.

```
auto x = Input();
auto y = Dense( 10, 28*28 )( x );
auto output = relu6( y );
```

6.1.2.94 replace_placeholder_with_expression()

```
template<Expression Ex, Place_Holder Ph, Expression Ey>
auto ceras::replace_placeholder_with_expression (
            Ex const & ex,
            Ph const & old_place_holder,
            Ey const & new_expression )
```

Replacing a place_holder with an expression.

Parameters

ex	Can be a unary operator, binary operator, variable, place_holder, a constant or a value
old_place_holder	An place holder in `ex`
new_expression	An expression that will replace `old_place_holder` in `ex`.

Returns

A expression inheriting the topology of `ex`, but with `old_place_holder` replaced by `new_expression`

6.1.2.95 Reshape()

```
auto ceras::Reshape (
            std::vector< unsigned long > const & new_shape,
            bool include_batch_flag = true ) [inline], [noexcept]
```

Reshapes inputs into the given shape.

6.1.2.96 run()

```
template<typename Operation >
auto ceras::run (
            Operation & op )
```

Run an expression.

Parameters

op	An expression.

Returns

The result of the expression.

6.1.2.97 selu()

```
template<Expression Ex>
auto ceras::selu (
            Ex const & ex ) [inline], [noexcept]
```

Scaled Exponential Linear Unit (SELU) activation function, an unary operator. If `x>0`, returns 1.0507 x; Otherwise, returns 1.67326∗1.0507∗(exp(x)-1)

Parameters

ex	An input operator

```
auto x = Input();
auto y = Dense( 10, 28*28 )( x );
auto output = selu( y );
```

6.1.2.98 sigmoid()

```
template<Expression Ex>
auto ceras::sigmoid (
            Ex const & ex ) [inline], [noexcept]
```

Sigmoid function, an unary operator. Returns `1 / (exp(-x) + 1)`.

Parameters

ex	An input operator.

```
auto x = Input();
auto y = Dense( 10, 28*28 )( x );
auto output = sigmoid( y );
```

6.1.2.99 silu()

```
template<Expression Ex>
auto ceras::silu (
            Ex const & ex )  [noexcept]
```

An alias name of activation swish.

6.1.2.100 soft_sign()

```
template<Expression Ex>
constexpr auto ceras::soft_sign (
            Ex const & ex )  [constexpr], [noexcept]
```

6.1.2.101 Softmax()

```
auto ceras::Softmax ( )  [inline], [noexcept]
```

Softmax activation function.

6.1.2.102 softmax()

```
template<Expression Ex>
constexpr auto ceras::softmax (
            Ex const & ex )  [constexpr], [noexcept]
```

Softmax activation function, an unary operator.

Parameters

ex	An input operator

```
auto x = Input();
auto y = Dense( 10, 28*28 )( x );
auto output = softmax( y );
```

6.1.2.103 softplus()

```
template<Expression Ex>
auto ceras::softplus (
            Ex const & ex )  [inline], [noexcept]
```

Softplus function, an unary operator. Returns `log(exp(x)+1)`.

Parameters

ex	An input operator

```
auto x = Input();
auto y = Dense( 10, 28*28 )( x );
auto output = softplus( y );
```

6.1.2.104 softsign()

```
template<Expression Ex>
auto ceras::softsign (
            Ex const & ex )  [inline], [noexcept]
```

Softsign function, an unary operator. Returns `x / (abs(x) + 1)`.

Parameters

ex	An input operator.

```
auto x = Input();
auto y = Dense( 10, 28*28 )( x );
auto output = softsign( y );
```

6.1.2.105 square()

```
template<Expression Ex>
constexpr auto ceras::square (
            Ex const & ex )  [constexpr], [noexcept]
```

Returns the square of the input

Parameters

ex	The input operator.

Returns

An instance of a unary_operator that evaluate the squared value of the input operator.

Example code:

```
auto e = variable<tensor<float>>{ /*...*/ };
auto square = square(e);
```

6.1.2.106 squared_loss()

```
template<Expression Lhs_Expression, Expression Rhs_Expression>
constexpr auto ceras::squared_loss (
            Lhs_Expression const & lhs_ex,
            Rhs_Expression const & rhs_ex ) [constexpr], [noexcept]
```

6.1.2.107 Subtract()

```
auto ceras::Subtract ( ) [inline], [noexcept]
```

Layer that subtracts two layers

Example usage:

```
auto input = Input(); // (16, )
auto x1 = Dense( 8, 16 )( input );
auto x2 = Dense( 8, 16 )( input );
auto x3 = Subtract()( x1, x2 ); // equivalent to 'x1 - x2'
auto m = model{ input, x3 };
```

6.1.2.108 sum_reduce()

```
template<Expression Ex>
constexpr auto ceras::sum_reduce (
            Ex const & ex ) [constexpr], [noexcept]
```

Sum up all elements, returns a scalar.

```
auto x = variable{ ... };
auto y = sum_reduce( x );
```

6.1.2.109 swish()

```
template<Expression Ex>
auto ceras::swish (
            Ex const & ex ) [noexcept]
```

Swish activation function.

Reference: Ramachandran, Prajit, Barret Zoph, and Quoc V. Le. "Searching for Activation Functions." ArXiv:1710.↩
05941 [Cs], October 16, 2017. http://arxiv.org/abs/1710.05941.

Parameters

ex	Input expression.

6.1.2.110 tank_shrink()

```
template<Expression Ex>
auto ceras::tank_shrink (
            Ex const & ex )  [noexcept]
```

Tank shrink function.

```
auto v = variable{ random<float>{ 3, 3 } };
auto c = tank_shrink( v );
```

6.1.2.111 unit_step()

```
template<Expression Ex>
constexpr auto ceras::unit_step (
            Ex const & ex )  [constexpr], [noexcept]
```

6.1.2.112 UpSampling2D()

```
auto ceras::UpSampling2D (
            unsigned long stride )  [inline], [noexcept]
```

Upsampling layer for 2D inputs.

6.1.3 Variable Documentation

6.1.3.1 Adadelta

```
auto ceras::Adadelta  [inline]
```

Initial value:

```
= []( auto ...  args )
    {
        return [=]<Expression Ex>( Ex& loss )
        {
            return adadelta{loss, args...};
        };
    }
```

6.1.3.2 Adagrad

auto ceras::Adagrad [inline]

Initial value:
```
= []( auto ...  args )
    {
        return [=]<Expression Ex>( Ex& loss )
        {
            return adagrad{loss, args...};
        };
    }
```

6.1.3.3 Adam

auto ceras::Adam [inline]

Initial value:
```
= []( auto ...  args )
    {
        return [=]<Expression Ex>( Ex& loss )
        {
            return adam{loss, args...};
        };
    }
```

6.1.3.4 BinaryCrossentropy

constexpr auto ceras::BinaryCrossentropy [inline], [static], [constexpr]

Initial value:
```
= []()
    {
        return []<Expression Ex >( Ex const& output )
        {
            return [=]<Place_Holder Ph>( Ph const& ground_truth )
            {
                return binary_cross_entropy_loss( ground_truth, output );
            };
        };
    }
```

6.1.3.5 BinaryCrossEntropy

constexpr auto ceras::BinaryCrossEntropy [inline], [static], [constexpr]

Initial value:
```
= []()
    {
        return BinaryCrossentropy();
    }
```

6.1.3.6 CategoricalCrossentropy

```
constexpr auto ceras::CategoricalCrossentropy [inline], [static], [constexpr]
```

Initial value:

```
= []<std::floating_point F=float>( F label_smoothing_factor = 0.0)
    {
        return [=]<Expression Ex >( Ex const& output )
        {
            return [=]<Place_Holder Ph>( Ph const& ground_truth )
            {
                return cross_entropy_loss( ground_truth, output, label_smoothing_factor );
            };
        };
    }
```

6.1.3.7 CategoricalCrossEntropy

```
constexpr auto ceras::CategoricalCrossEntropy [inline], [static], [constexpr]
```

Initial value:

```
= []<std::floating_point F=float>(F label_smoothing_factor = 0.0)
    {
        return CategoricalCrossentropy(label_smoothing_factor);
    }
```

6.1.3.8 Hinge

```
constexpr auto ceras::Hinge [inline], [static], [constexpr]
```

Initial value:

```
= []()
    {
        return []<Expression Ex >( Ex const& output )
        {
            return [=]<Place_Holder Ph>( Ph const& ground_truth )
            {
                return hinge_loss( ground_truth, output );
            };
        };
    }
```

6.1.3.9 is_binary_operator_v

```
template<class T >
constexpr bool ceras::is_binary_operator_v = is_binary_operator<T>::value [inline], [constexpr]
```

If T is an instance of a binary_operator, the constant value equals to `true`. Otherwise this value is `false`.

6.1.3.10 is_complex_v

```
template<typename T >
constexpr bool ceras::is_complex_v = is_complex<T>::value [constexpr]
```

6.1.3.11 is_constant_v

```
template<class T >
constexpr bool ceras::is_constant_v = is_constant<T>::value [inline], [constexpr]
```

6.1.3.12 is_place_holder_v

```
template<class T >
constexpr bool ceras::is_place_holder_v = is_place_holder<T>::value [inline], [constexpr]
```

6.1.3.13 is_tensor_v

```
template<class T >
constexpr bool ceras::is_tensor_v = is_tensor<T>::value [inline], [constexpr]
```

6.1.3.14 is_unary_operator_v

```
template<class T >
constexpr bool ceras::is_unary_operator_v = is_unary_operator<T>::value [inline], [constexpr]
```

If T is an instance of a unary_operator, the constant value equals to `true`. `false` otherwise.

6.1.3.15 is_value_v

```
template<class T >
constexpr bool ceras::is_value_v = is_value<T>::value [inline], [constexpr]
```

6.1.3.16 is_variable_v

```
template<class T >
constexpr bool ceras::is_variable_v = is_variable<T>::value [inline], [constexpr]
```

6.1.3.17 MAE

```
constexpr auto ceras::MAE [inline], [static], [constexpr]
```

Initial value:

```
= [] ()
    {
        return MeanAbsoluteError();
    }
```

An alias name of function MeanAbsoluteError.

6.1.3.18 MeanAbsoluteError

```
constexpr auto ceras::MeanAbsoluteError [inline], [static], [constexpr]
```

Initial value:

```
= [] ()
    {
        return []<Expression Ex >( Ex const& output )
        {
            return [=]<Place_Holder Ph>( Ph const& ground_truth )
            {
                return mean_absolute_error( ground_truth, output );
            };
        };
    }
```

Computes the mean of absolute errors between labels and predictions.

```
auto input = place_holder<tensor<float>>{};
auto v = variable<tensor<float»{ ones<float>({12, 34}) };
auto output = input * v;
auto m = model{ input, output };
auto cm = m.compile( MeanAbsoluteError(), Adam(128/*batch size*/, 0.01f/*learning rate*/) );
```

see also mean_absolute_error

6.1.3.19 MeanSquaredError

```
constexpr auto ceras::MeanSquaredError [inline], [static], [constexpr]
```

Initial value:

```
= [] ()
    {
        return []<Expression Ex >( Ex const& output )
        {
            return [=]<Place_Holder Ph>( Ph const& ground_truth )
            {
                return mean_squared_error( ground_truth, output );
            };
        };
    }
```

Computes the mean of squares of errors between labels and predictions.

```
auto input = place_holder<tensor<float>>{};
auto v = variable<tensor<float»{ ones<float>({12, 34}) };
auto output = input * v;
auto m = model{ input, output };
auto cm = m.compile( MeanSquareError(), Adam(128/*batch size*/, 0.01f/*learning rate*/) );
```

see also mean_squared_error

6.1.3.20 MSE

```
constexpr auto ceras::MSE [inline], [static], [constexpr]
```

Initial value:
```
= []()
    {
        return MeanSquaredError();
    }
```

An alias name of function MeanSquaredError.

6.1.3.21 random_generator

```
std::mt19937 ceras::random_generator {random_seed} [static]
```

6.1.3.22 random_seed

```
unsigned long ceras::random_seed = std::chrono::system_clock::now().time_since_epoch().count()
[static]
```

Random seed for the tensor library.

To reproduce the result involving random variates such as `rand`, `normal`, `poisson`, it is necessary to fix the random seed by

```
random_seed=42;
```

6.1.3.23 RMSprop

```
auto ceras::RMSprop [inline]
```

Initial value:
```
= []( auto ...  args )
    {
        return [=]<Expression Ex>( Ex& loss )
        {
            return rmsprop{loss, args...};
        };
    }
```

6.1.3.24 SGD

```
auto ceras::SGD [inline]
```

Initial value:
```
= []( auto ...  args )
    {
        return [=]<Expression Ex>( Ex& loss )
        {
            return sgd{loss, args...};
        };
    }
```

6.2 ceras::ceras_private Namespace Reference

Classes

- struct session

6.3 ceras::dataset Namespace Reference

Namespaces

- namespace fashion_mnist
- namespace mnist

6.4 ceras::dataset::fashion_mnist Namespace Reference

Functions

- auto load_data (std::string const &path=std::string{"./dataset/fashion_mnist"})

6.4.1 Function Documentation

6.4.1.1 load_data()

```
auto ceras::dataset::fashion_mnist::load_data (
            std::string const & path = std::string{"./dataset/fashion_mnist"} )  [inline]
```

Loads the fashion-MNIST dataset.

Parameters

path	Path where to cache the dataset locally. Default to "./dataset/fashion_mnist", should be updated if running the program somewhere else.

Returns

A tuple of 4 tensors: x_train, y_train, x_test, y_test. x_train, x_test: uint8 arrays of grayscale image data with shapes (num_samples, 28, 28). y_train, y_test: uint8 tensor of digit labels (integers in range 0-9) with shapes (num_samples, 10). Note: for digit 0, the corresponding array is `[[1, 0, 0, 0, 0, 0, 0, 0, 0, 0]]`.

Label Description 0 T-shirt/top 1 Trouser 2 Pullover 3 Dress 4 Coat 5 Sandal 6 Shirt 7 Sneaker 8 Bag 9 Ankle boot

Example usage:
```
auto const& [x_train, y_train, x_test, y_test] =
      ceras::dataset::mnist::load_data("/home/feng/dataset/fashion_mnist");
```

The copyright for Fashion-MNIST is held by Zalando SE. Fashion-MNIST is licensed under the MIT license.

6.5 ceras::dataset::mnist Namespace Reference

Functions

- auto load_data (std::string const &path=std::string{"./dataset/mnist"})

6.5.1 Function Documentation

6.5.1.1 load_data()

```
auto ceras::dataset::mnist::load_data (
            std::string const & path = std::string{"./dataset/mnist"} )  [inline]
```

Loads the MNIST dataset.

Parameters

path	Path where to cache the dataset locally. Default to "./dataset/mnist", should be updated if running the program somewhere else.

Returns

A tuple of 4 tensors: x_train, y_train, x_test, y_test. x_train, x_test: uint8 arrays of grayscale image data with shapes (num_samples, 28, 28). y_train, y_test: uint8 tensor of digit labels (integers in range 0-9) with shapes (num_samples, 10). Note: for digit 0, the corresponding array is `[[1, 0, 0, 0, 0, 0, 0, 0, 0, 0]]`.

Example usage:
```
auto const& [x_train, y_train, x_test, y_test] =
      ceras::dataset::mnist::load_data("/home/feng/dataset/mnist");
```

Yann LeCun and Corinna Cortes hold the copyright of MNIST dataset, which is available under the terms of the Creative Commons Attribution-Share Alike 3.0 license.

Chapter 7

Concept Documentation

7.1 ceras::Binary_Operator Concept Reference

A type that represents a binary operator.

`#include <operation.hpp>`

7.1.1 Concept definition

```
template<typename T>
concept ceras::Binary_Operator =  is_binary_operator_v<T>
```

7.1.2 Detailed Description

A type that represents a binary operator.

<>

7.2 ceras::Complex Concept Reference

A type that represents a complex expression.

`#include <complex_operator.hpp>`

7.2.1 Concept definition

```
template<typename T>
concept ceras::Complex =  is_complex_v<T>
```

7.2.2 Detailed Description

A type that represents a complex expression.

7.3 ceras::Constant Concept Reference

```
#include <constant.hpp>
```

7.3.1 Concept definition

```
template<typename T>
concept ceras::Constant =  is_constant_v<T>
```

7.4 ceras::Expression Concept Reference

A type that represents a unary operator, a binary operator, a variable, a place_holder, a constant or a value.

```
#include <operation.hpp>
```

7.4.1 Concept definition

```
template<typename T>
concept ceras::Expression =  Operator<T> || Variable<T> || Place_Holder<T> || Constant<T> || Value<T>
```

7.4.2 Detailed Description

A type that represents a unary operator, a binary operator, a variable, a place_holder, a constant or a value.

<>

7.5 ceras::Operator Concept Reference

A type that represents an unary or a binary operator.

```
#include <operation.hpp>
```

7.5.1 Concept definition

```
template<typename T>
concept ceras::Operator =  Unary_Operator<T> || Binary_Operator<T>
```

7.5.2 Detailed Description

A type that represents an unary or a binary operator.

<>

7.6 ceras::Place_Holder Concept Reference

```
#include <place_holder.hpp>
```

7.6.1 Concept definition

```
template<typename T>
concept ceras::Place_Holder =  is_place_holder_v<T>
```

7.7 ceras::Tensor Concept Reference

```
#include <tensor.hpp>
```

7.7.1 Concept definition

```
template<typename T>
concept ceras::Tensor =  is_tensor_v<T>
```

7.8 ceras::Unary_Operator Concept Reference

A type that represents an unary operator.

```
#include <operation.hpp>
```

7.8.1 Concept definition

```
template<typename T>
concept ceras::Unary_Operator =  is_unary_operator_v<T>
```

7.8.2 Detailed Description

A type that represents an unary operator.

<>

7.9 ceras::Value Concept Reference

```
#include <value.hpp>
```

7.9.1 Concept definition

```
template<typename T>
concept ceras::Value =  is_value_v<T>
```

7.10 ceras::Variable Concept Reference

```
#include <variable.hpp>
```

7.10.1 Concept definition

```
template<typename T>
concept ceras::Variable =  is_variable_v<T>
```

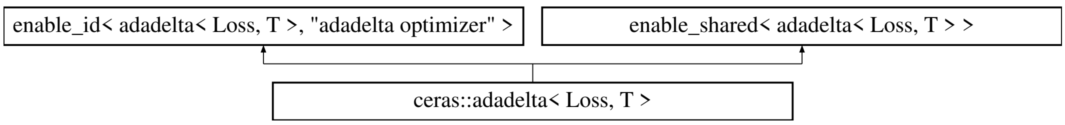
Chapter 8

Class Documentation

8.1 ceras::adadelta< Loss, T > Struct Template Reference

```
#include <optimizer.hpp>
```

Inheritance diagram for ceras::adadelta< Loss, T >:



Public Types

- typedef tensor< T > tensor_type

Public Member Functions

- adadelta (Loss &loss, std::size_t batch_size, T rho=0.9) noexcept
- void forward ()

Public Attributes

- Loss & loss_
- T rho_
- T learning_rate_
- unsigned long iterations_

8.1.1 Member Typedef Documentation

8.1.1.1 tensor_type

```
template<typename Loss , typename T >
typedef tensor< T > ceras::adadelta< Loss, T >::tensor_type
```

8.1.2 Constructor & Destructor Documentation

8.1.2.1 adadelta()

```
template<typename Loss , typename T >
ceras::adadelta< Loss, T >::adadelta (
            Loss & loss,
            std::size_t batch_size,
            T rho = 0.9 )  [inline], [noexcept]
```

8.1.3 Member Function Documentation

8.1.3.1 forward()

```
template<typename Loss , typename T >
void ceras::adadelta< Loss, T >::forward ( )  [inline]
```

8.1.4 Member Data Documentation

8.1.4.1 iterations_

```
template<typename Loss , typename T >
unsigned long ceras::adadelta< Loss, T >::iterations_
```

8.1.4.2 learning_rate_

```
template<typename Loss , typename T >
T ceras::adadelta< Loss, T >::learning_rate_
```

8.1.4.3 loss_

```
template<typename Loss , typename T >
Loss& ceras::adadelta< Loss, T >::loss_
```

8.1.4.4 rho_

```
template<typename Loss , typename T >
T ceras::adadelta< Loss, T >::rho_
```

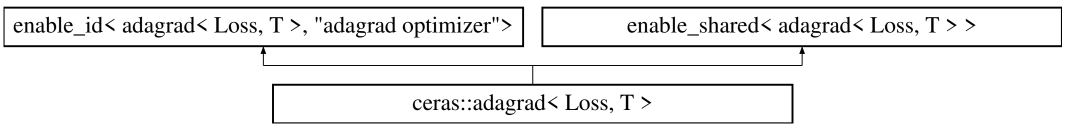
The documentation for this struct was generated from the following file:

- /home/feng/workspace/github.repo/ceras/include/optimizer.hpp

8.2 ceras::adagrad< Loss, T > Struct Template Reference

```
#include <optimizer.hpp>
```

Inheritance diagram for ceras::adagrad< Loss, T >:



Public Types

- typedef tensor< T > tensor_type

Public Member Functions

- adagrad (Loss &loss, std::size_t batch_size, T learning_rate=1.0e-1, T decay=0.0) noexcept
- void forward ()

Public Attributes

- Loss & loss_
- T learning_rate_
- T decay_
- unsigned long iterations_

8.2.1 Member Typedef Documentation

8.2.1.1 tensor_type

```
template<typename Loss , typename T >
typedef tensor< T > ceras::adagrad< Loss, T >::tensor_type
```

8.2.2 Constructor & Destructor Documentation

8.2.2.1 adagrad()

```
template<typename Loss , typename T >
ceras::adagrad< Loss, T >::adagrad (
            Loss & loss,
            std::size_t batch_size,
            T learning_rate = 1.0e-1,
            T decay = 0.0 )  [inline], [noexcept]
```

8.2.3 Member Function Documentation

8.2.3.1 forward()

```
template<typename Loss , typename T >
void ceras::adagrad< Loss, T >::forward ( )  [inline]
```

8.2.4 Member Data Documentation

8.2.4.1 decay_

```
template<typename Loss , typename T >
T ceras::adagrad< Loss, T >::decay_
```

8.2.4.2 iterations_

```
template<typename Loss , typename T >
unsigned long ceras::adagrad< Loss, T >::iterations_
```

8.2.4.3 learning_rate_

```
template<typename Loss , typename T >
T ceras::adagrad< Loss, T >::learning_rate_
```

8.2.4.4 loss_

```
template<typename Loss , typename T >
Loss& ceras::adagrad< Loss, T >::loss_
```

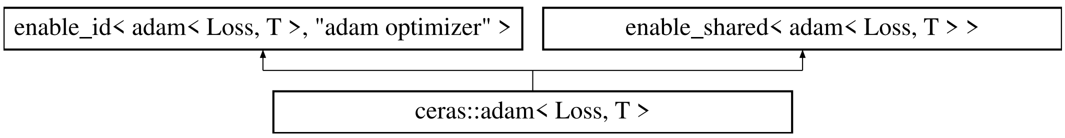
The documentation for this struct was generated from the following file:

- /home/feng/workspace/github.repo/ceras/include/optimizer.hpp

8.3 ceras::adam< Loss, T > Struct Template Reference

```
#include <optimizer.hpp>
```

Inheritance diagram for ceras::adam< Loss, T >:



Public Types

- typedef tensor< T > tensor_type

Public Member Functions

- adam (Loss &loss, std::size_t batch_size, T learning_rate=1.0e-1, T beta_1=0.9, T beta_2=0.999, bool amsgrad=false) noexcept
- void forward ()

Public Attributes

- Loss & loss_
- T learning_rate_
- T beta_1_
- T beta_2_
- bool amsgrad_
- unsigned long iterations_

8.3.1 Member Typedef Documentation

8.3.1.1 tensor_type

```
template<typename Loss , typename T >
typedef tensor< T > ceras::adam< Loss, T >::tensor_type
```

8.3.2 Constructor & Destructor Documentation

8.3.2.1 adam()

```
template<typename Loss , typename T >
ceras::adam< Loss, T >::adam (
            Loss & loss,
            std::size_t batch_size,
            T learning_rate = 1.0e-1,
            T beta_1 = 0.9,
            T beta_2 = 0.999,
            bool amsgrad = false )  [inline], [noexcept]
```

8.3.3 Member Function Documentation

8.3.3.1 forward()

```
template<typename Loss , typename T >
void ceras::adam< Loss, T >::forward ( )  [inline]
```

8.3.4 Member Data Documentation

8.3.4.1 amsgrad_

```
template<typename Loss , typename T >
bool ceras::adam< Loss, T >::amsgrad_
```

8.3.4.2 beta_1_

```
template<typename Loss , typename T >
T ceras::adam< Loss, T >::beta_1_
```

8.3.4.3 beta_2_

```
template<typename Loss , typename T >
T ceras::adam< Loss, T >::beta_2_
```

8.3.4.4 iterations_

```
template<typename Loss , typename T >
unsigned long ceras::adam< Loss, T >::iterations_
```

8.3.4.5 learning_rate_

```
template<typename Loss , typename T >
T ceras::adam< Loss, T >::learning_rate_
```

8.3.4.6 loss_

```
template<typename Loss , typename T >
Loss& ceras::adam< Loss, T >::loss_
```

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

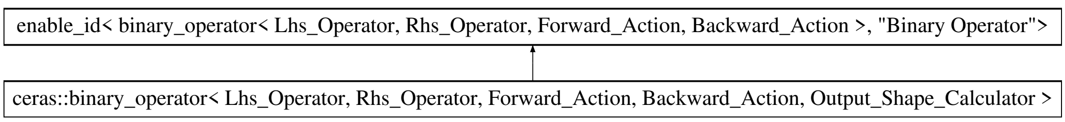
- /home/feng/workspace/github.repo/ceras/include/optimizer.hpp

8.4 ceras::binary_operator< Lhs_Operator, Rhs_Operator, Forward_Action, Backward_Action, Output_Shape_Calculator > Struct Template Reference

A binary operator is composed of a.) a left-side input expression, b.) a right-side input expression, c.) a forward action and d.) a backward action.

```
#include <operation.hpp>
```

Inheritance diagram for ceras::binary_operator< Lhs_Operator, Rhs_Operator, Forward_Action, Backward_Action, Output_Shape_Calculator >:



Public Types

- typedef tensor_deduction< Lhs_Operator, Rhs_Operator >::tensor_type tensor_type

Public Member Functions

- binary_operator (Lhs_Operator const &lhs_op, Rhs_Operator const &rhs_op, Forward_Action const &forward_action, Backward_Action const &backward_action, Output_Shape_Calculator const &output↩ _shape_calculator) noexcept
- auto forward ()
- void backward (tensor_type const &grad)

  Backward action, grad back-propagated.
- std::vector< unsigned long > shape () const noexcept

  Calculate the output shape.

Public Attributes

- Lhs_Operator lhs_op_
- Rhs_Operator rhs_op_
- Forward_Action forward_action_
- Backward_Action backward_action_
- Output_Shape_Calculator output_shape_calculator_
- tensor_type lhs_input_data_
- tensor_type rhs_input_data_
- tensor_type output_data_

8.4.1 Detailed Description

template< typename Lhs_Operator, typename Rhs_Operator, typename Forward_Action, typename Backward_Action, typename Output_Shape_Calculator = identity_output_shape_calculator>
struct ceras::binary_operator< Lhs_Operator, Rhs_Operator, Forward_Action, Backward_Action, Output_Shape_Calculator >

A binary operator is composed of a.) a left-side input expression, b.) a right-side input expression, c.) a forward action and d.) a backward action.

8.4.2 Member Typedef Documentation

8.4.2.1 tensor_type

```
template<typename Lhs_Operator , typename Rhs_Operator , typename Forward_Action , typename
Backward_Action , typename Output_Shape_Calculator = identity_output_shape_calculator>
typedef tensor_deduction<Lhs_Operator,Rhs_Operator>::tensor_type ceras::binary_operator<
Lhs_Operator, Rhs_Operator, Forward_Action, Backward_Action, Output_Shape_Calculator >::tensor_type
```

8.4.3 Constructor & Destructor Documentation

8.4.3.1 binary_operator()

```
template<typename Lhs_Operator , typename Rhs_Operator , typename Forward_Action , typename
Backward_Action , typename Output_Shape_Calculator = identity_output_shape_calculator>
ceras::binary_operator< Lhs_Operator, Rhs_Operator, Forward_Action, Backward_Action, Output_↩
Shape_Calculator >::binary_operator (
            Lhs_Operator const & lhs_op,
            Rhs_Operator const & rhs_op,
            Forward_Action const & forward_action,
            Backward_Action const & backward_action,
            Output_Shape_Calculator const & output_shape_calculator )  [inline], [noexcept]
```

8.4.4 Member Function Documentation

8.4.4.1 backward()

```
template<typename Lhs_Operator , typename Rhs_Operator , typename Forward_Action , typename
Backward_Action , typename Output_Shape_Calculator = identity_output_shape_calculator>
void ceras::binary_operator< Lhs_Operator, Rhs_Operator, Forward_Action, Backward_Action,
Output_Shape_Calculator >::backward (
            tensor_type const & grad )  [inline]
```

Backward action, grad back-propagated.

8.4.4.2 forward()

```
template<typename Lhs_Operator , typename Rhs_Operator , typename Forward_Action , typename
Backward_Action , typename Output_Shape_Calculator = identity_output_shape_calculator>
auto ceras::binary_operator< Lhs_Operator, Rhs_Operator, Forward_Action, Backward_Action,
Output_Shape_Calculator >::forward ( )  [inline]
```

8.4.4.3 shape()

```
template<typename Lhs_Operator , typename Rhs_Operator , typename Forward_Action , typename
Backward_Action , typename Output_Shape_Calculator = identity_output_shape_calculator>
std::vector< unsigned long > ceras::binary_operator< Lhs_Operator, Rhs_Operator, Forward_↩
Action, Backward_Action, Output_Shape_Calculator >::shape ( ) const  [inline], [noexcept]
```

Calculate the output shape.

8.4.5 Member Data Documentation

8.4.5.1 backward_action_

template<typename Lhs_Operator , typename Rhs_Operator , typename Forward_Action , typename Backward_Action , typename Output_Shape_Calculator = identity_output_shape_calculator>
Backward_Action ceras::binary_operator< Lhs_Operator, Rhs_Operator, Forward_Action, Backward↩
_Action, Output_Shape_Calculator >::backward_action_

8.4.5.2 forward_action_

template<typename Lhs_Operator , typename Rhs_Operator , typename Forward_Action , typename Backward_Action , typename Output_Shape_Calculator = identity_output_shape_calculator>
Forward_Action ceras::binary_operator< Lhs_Operator, Rhs_Operator, Forward_Action, Backward_↩
Action, Output_Shape_Calculator >::forward_action_

8.4.5.3 lhs_input_data_

template<typename Lhs_Operator , typename Rhs_Operator , typename Forward_Action , typename Backward_Action , typename Output_Shape_Calculator = identity_output_shape_calculator>
tensor_type ceras::binary_operator< Lhs_Operator, Rhs_Operator, Forward_Action, Backward_↩
Action, Output_Shape_Calculator >::lhs_input_data_

8.4.5.4 lhs_op_

template<typename Lhs_Operator , typename Rhs_Operator , typename Forward_Action , typename Backward_Action , typename Output_Shape_Calculator = identity_output_shape_calculator>
Lhs_Operator ceras::binary_operator< Lhs_Operator, Rhs_Operator, Forward_Action, Backward_↩
Action, Output_Shape_Calculator >::lhs_op_

8.4.5.5 output_data_

template<typename Lhs_Operator , typename Rhs_Operator , typename Forward_Action , typename Backward_Action , typename Output_Shape_Calculator = identity_output_shape_calculator>
tensor_type ceras::binary_operator< Lhs_Operator, Rhs_Operator, Forward_Action, Backward_↩
Action, Output_Shape_Calculator >::output_data_

8.4.5.6 output_shape_calculator_

```
template<typename Lhs_Operator , typename Rhs_Operator , typename Forward_Action , typename
Backward_Action , typename Output_Shape_Calculator = identity_output_shape_calculator>
Output_Shape_Calculator ceras::binary_operator< Lhs_Operator, Rhs_Operator, Forward_Action,
Backward_Action, Output_Shape_Calculator >::output_shape_calculator_
```

8.4.5.7 rhs_input_data_

```
template<typename Lhs_Operator , typename Rhs_Operator , typename Forward_Action , typename
Backward_Action , typename Output_Shape_Calculator = identity_output_shape_calculator>
tensor_type ceras::binary_operator< Lhs_Operator, Rhs_Operator, Forward_Action, Backward_↩
Action, Output_Shape_Calculator >::rhs_input_data_
```

8.4.5.8 rhs_op_

```
template<typename Lhs_Operator , typename Rhs_Operator , typename Forward_Action , typename
Backward_Action , typename Output_Shape_Calculator = identity_output_shape_calculator>
Rhs_Operator ceras::binary_operator< Lhs_Operator, Rhs_Operator, Forward_Action, Backward_↩
Action, Output_Shape_Calculator >::rhs_op_
```

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

- /home/feng/workspace/github.repo/ceras/include/operation.hpp

8.5 ceras::compiled_model< Model, Optimizer, Loss > Struct Template Reference

```
#include <model.hpp>
```

Public Types

- typedef Model::input_layer_type io_layer_type
- typedef decltype(std::declval< Optimizer >()(std::declval< Loss & >())) optimizer_type

Public Member Functions

- compiled_model (Model const &m, io_layer_type const &input_place_holder, io_layer_type const &ground↩ _truth_place_holder, Loss const &loss, Optimizer const &optimizer)
- template< Tensor Tsor >
  auto evaluate (Tsor const &inputs, Tsor const &outputs, unsigned long batch_size=32)
- template< Tensor Tsor >
  auto fit (Tsor const &inputs, Tsor const &outputs, unsigned long batch_size, unsigned long epoch=1, int verbose=0, double validation_split=0.0)
- template< Tensor Tsor >
  auto train_on_batch (Tsor const &input, Tsor const &output)
- template< Tensor Tsor >
  auto predict (Tsor const &input_tensor)
- template< Expression Exp >
  auto operator() (Exp const &ex) const noexcept
- void trainable (bool t)

Public Attributes

- Model model_
- io_layer_type input_place_holder_
- io_layer_type ground_truth_place_holder_
- Loss loss_
- Optimizer optimizer_
- optimizer_type compiled_optimizer_

8.5.1 Member Typedef Documentation

8.5.1.1 io_layer_type

```
template<typename Model , typename Optimizer , typename Loss >
typedef Model::input_layer_type ceras::compiled_model< Model, Optimizer, Loss >::io_layer_type
```

8.5.1.2 optimizer_type

```
template<typename Model , typename Optimizer , typename Loss >
typedef decltype(std::declval<Optimizer>()(std::declval<Loss&>())) ceras::compiled_model<
Model, Optimizer, Loss >::optimizer_type
```

8.5.2 Constructor & Destructor Documentation

8.5.2.1 compiled_model()

```
template<typename Model , typename Optimizer , typename Loss >
ceras::compiled_model< Model, Optimizer, Loss >::compiled_model (
            Model const & m,
            io_layer_type const & input_place_holder,
            io_layer_type const & ground_truth_place_holder,
            Loss const & loss,
            Optimizer const & optimizer )  [inline]
```

8.5.3 Member Function Documentation

8.5.3.1 evaluate()

```
template<typename Model , typename Optimizer , typename Loss >
template<Tensor Tsor>
auto ceras::compiled_model< Model, Optimizer, Loss >::evaluate (
            Tsor const & inputs,
            Tsor const & outputs,
            unsigned long batch_size = 32 )  [inline]
```

Calculate the loss for the model in test model.

Parameters

inputs	Input data. A tensor of shape (samples, input_shape).
outputs	Output data. A tensor of shape (samples, output_shape).
batch_size	Number of samples per batch of computation. Default to 32.

Returns

Test loss. A scalar.

8.5.3.2 fit()

```
template<typename Model , typename Optimizer , typename Loss >
template<Tensor Tsor>
auto ceras::compiled_model< Model, Optimizer, Loss >::fit (
            Tsor const & inputs,
            Tsor const & outputs,
            unsigned long batch_size,
            unsigned long epoch = 1,
            int verbose = 0,
            double validation_split = 0.0 )  [inline]
```

Train the model on the selected dataset for a fixed numbers of epoches.

Parameters

inputs	Input data. A tensor of shape (samples, input_shape).
outputs	Input data. A tensor of shape (samples, output_shape).
batch_size	Number of samples per gradient update. Should agree with the batch size in the optimizer.
epoch	Number of epoches to train the dataset.
verbose	Verbosity mode. 0 for slient. 1 for one line per epoch.
validation_split	Fraction of the training data that will be used for validation. A floating number in range [0, 1].

Returns

A tuple of two vectors. The first vector gives the historical errors on the training data. The second vector gives the historical errors on the validation data.

Example:

```
model m{ ... };
auto cm = m.compile( ... );
tensor<float> inputs, outputs;
//...
unsigned long batch_size = 32;
unsigned long epoch = 10;
int verbose = 1;
double validation_split = 0.2;
auto errors = cm.fit( inputs, outputs, batch_size, epoch, verbose, validation_split );
```

8.5.3.3 operator()()

```
template<typename Model , typename Optimizer , typename Loss >
template<Expression Exp>
auto ceras::compiled_model< Model, Optimizer, Loss >::operator() (
            Exp const & ex ) const  [inline], [noexcept]
```

8.5.3.4 predict()

```
template<typename Model , typename Optimizer , typename Loss >
template<Tensor Tsor>
auto ceras::compiled_model< Model, Optimizer, Loss >::predict (
            Tsor const & input_tensor )  [inline]
```

8.5.3.5 train_on_batch()

```
template<typename Model , typename Optimizer , typename Loss >
template<Tensor Tsor>
auto ceras::compiled_model< Model, Optimizer, Loss >::train_on_batch (
            Tsor const & input,
            Tsor const & output )  [inline]
```

Running a single updated on a single batch of data.

Parameters

input	The input data to train the model. A tensor of shape (batch_size, input_shape).
output	The output data to train the model. A tensor of shape (batch_size, output_shape).

Returns

Training loss. A scalar.

Example code:

```
auto m = model{ ...  };
auto cm = m.compile( ...  );
for ( auto idx :  range( 1024 ) )
{
    auto x = ...; // get batch input
    auto y = ...; // get batch output
    cm.train_on_batch( x, y );
}
```

8.5.3.6 trainable()

```
template<typename Model , typename Optimizer , typename Loss >
void ceras::compiled_model< Model, Optimizer, Loss >::trainable (
            bool t )  [inline]
```

8.5.4 Member Data Documentation

8.5.4.1 compiled_optimizer_

```
template<typename Model , typename Optimizer , typename Loss >
optimizer_type ceras::compiled_model< Model, Optimizer, Loss >::compiled_optimizer_
```

8.5.4.2 ground_truth_place_holder_

```
template<typename Model , typename Optimizer , typename Loss >
io_layer_type ceras::compiled_model< Model, Optimizer, Loss >::ground_truth_place_holder_
```

8.5.4.3 input_place_holder_

```
template<typename Model , typename Optimizer , typename Loss >
io_layer_type ceras::compiled_model< Model, Optimizer, Loss >::input_place_holder_
```

8.5.4.4 loss_

```
template<typename Model , typename Optimizer , typename Loss >
Loss ceras::compiled_model< Model, Optimizer, Loss >::loss_
```

8.5.4.5 model_

```
template<typename Model , typename Optimizer , typename Loss >
Model ceras::compiled_model< Model, Optimizer, Loss >::model_
```

8.5.4.6 optimizer_

```
template<typename Model , typename Optimizer , typename Loss >
Optimizer ceras::compiled_model< Model, Optimizer, Loss >::optimizer_
```

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

- /home/feng/workspace/github.repo/ceras/include/model.hpp

8.6 ceras::complex< Real_Ex, Imag_Ex > Struct Template Reference

```
#include <complex_operator.hpp>
```

Public Attributes

- Real_Ex real_
- Imag_Ex imag_

8.6.1 Member Data Documentation

8.6.1.1 imag_

```
template<Expression Real_Ex, Expression Imag_Ex>
Imag_Ex ceras::complex< Real_Ex, Imag_Ex >::imag_
```

8.6.1.2 real_

```
template<Expression Real_Ex, Expression Imag_Ex>
Real_Ex ceras::complex< Real_Ex, Imag_Ex >::real_
```

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

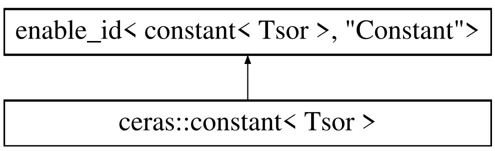
- /home/feng/workspace/github.repo/ceras/include/complex_operator.hpp

8.7 ceras::constant< Tsor > Struct Template Reference

Creates a constant expression from a tensor-like object.

```
#include <constant.hpp>
```

Inheritance diagram for ceras::constant< Tsor >:



Public Types

- typedef Tsor tensor_type

Public Member Functions

- constant (tensor_type const &data)
- void backward (auto) const
- tensor_type forward () const
- auto shape () const

Public Attributes

- tensor_type data_

8.7.1 Detailed Description

template<Tensor Tsor>
struct ceras::constant< Tsor >

Creates a constant expression from a tensor-like object.

```
auto c = constant{ zeros<float>( {3, 3, 3} ) };
```

8.7.2 Member Typedef Documentation

8.7.2.1 tensor_type

```
template<Tensor Tsor>
typedef Tsor ceras::constant< Tsor >::tensor_type
```

8.7.3 Constructor & Destructor Documentation

8.7.3.1 constant()

```
template<Tensor Tsor>
ceras::constant< Tsor >::constant (
              tensor_type const & data )   [inline]
```

8.7.4 Member Function Documentation

8.7.4.1 backward()

```
template<Tensor Tsor>
void ceras::constant< Tsor >::backward (
            auto ) const [inline]
```

8.7.4.2 forward()

```
template<Tensor Tsor>
tensor_type ceras::constant< Tsor >::forward ( ) const [inline]
```

8.7.4.3 shape()

```
template<Tensor Tsor>
auto ceras::constant< Tsor >::shape ( ) const [inline]
```

8.7.5 Member Data Documentation

8.7.5.1 data_

```
template<Tensor Tsor>
tensor_type ceras::constant< Tsor >::data_
```

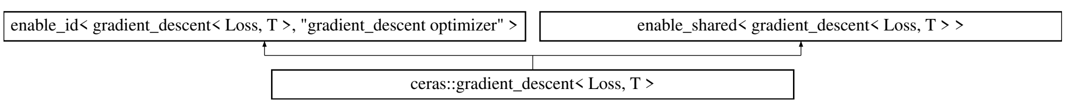
The documentation for this struct was generated from the following file:

- /home/feng/workspace/github.repo/ceras/include/constant.hpp

8.8 ceras::gradient_descent< Loss, T > Struct Template Reference

```
#include <optimizer.hpp>
```

Inheritance diagram for ceras::gradient_descent< Loss, T >:



Public Types

- typedef tensor< T > tensor_type

Public Member Functions

- gradient_descent (Loss &loss, std::size_t batch_size, T learning_rate=1.0e-3, T momentum=0.0) noexcept
- void forward ()

Public Attributes

- Loss & loss_
- T learning_rate_
- T momentum_

8.8.1 Member Typedef Documentation

8.8.1.1 tensor_type

```
template<typename Loss , typename T >
typedef tensor< T > ceras::gradient_descent< Loss, T >::tensor_type
```

8.8.2 Constructor & Destructor Documentation

8.8.2.1 gradient_descent()

```
template<typename Loss , typename T >
ceras::gradient_descent< Loss, T >::gradient_descent (
            Loss & loss,
            std::size_t batch_size,
            T learning_rate = 1.0e-3,
            T momentum = 0.0 ) [inline], [noexcept]
```

8.8.3 Member Function Documentation

8.8.3.1 forward()

```
template<typename Loss , typename T >
void ceras::gradient_descent< Loss, T >::forward ( ) [inline]
```

8.8.4 Member Data Documentation

8.8.4.1 learning_rate_

```
template<typename Loss , typename T >
T ceras::gradient_descent< Loss, T >::learning_rate_
```

8.8.4.2 loss_

```
template<typename Loss , typename T >
Loss& ceras::gradient_descent< Loss, T >::loss_
```

8.8.4.3 momentum_

```
template<typename Loss , typename T >
T ceras::gradient_descent< Loss, T >::momentum_
```

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

- /home/feng/workspace/github.repo/ceras/include/optimizer.hpp

8.9 ceras::identity_output_shape_calculator Struct Reference

The default identity output shape calculator for unary/binary operators. Should be overrided for some special operators.

```
#include <operation.hpp>
```

Public Member Functions

- std::vector< unsigned long > operator() (std::vector< unsigned long > const &input_shape) const noexcept
- std::vector< unsigned long > operator() (std::vector< unsigned long > const &lhs_input_shape, std::vector< unsigned long > const &rhs_input_shape) const noexcept
- std::vector< unsigned long > operator() () const noexcept

8.9.1 Detailed Description

The default identity output shape calculator for unary/binary operators. Should be overrided for some special operators.

8.9.2 Member Function Documentation

8.9.2.1 operator()() [1/3]

```
std::vector< unsigned long > ceras::identity_output_shape_calculator::operator() ( ) const
[inline], [noexcept]
```

8.9.2.2 operator()() [2/3]

```
std::vector< unsigned long > ceras::identity_output_shape_calculator::operator() (
            std::vector< unsigned long > const & input_shape ) const  [inline], [noexcept]
```

8.9.2.3 operator()() [3/3]

```
std::vector< unsigned long > ceras::identity_output_shape_calculator::operator() (
            std::vector< unsigned long > const & lhs_input_shape,
            std::vector< unsigned long > const & rhs_input_shape ) const  [inline], [noexcept]
```

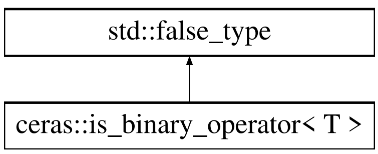
The documentation for this struct was generated from the following file:

- /home/feng/workspace/github.repo/ceras/include/operation.hpp

8.10 ceras::is_binary_operator< T > Struct Template Reference

```
#include <operation.hpp>
```

Inheritance diagram for ceras::is_binary_operator< T >:



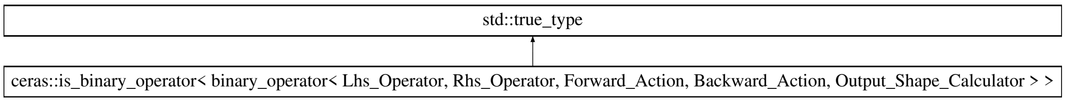
The documentation for this struct was generated from the following file:

- /home/feng/workspace/github.repo/ceras/include/operation.hpp

8.11 ceras::is_binary_operator< binary_operator< Lhs_Operator, Rhs_Operator, Forward_Action, Backward_Action, Output_Shape_Calculator > > Struct Template Reference

```
#include <operation.hpp>
```

Inheritance diagram for ceras::is_binary_operator< binary_operator< Lhs_Operator, Rhs_Operator, Forward_Action, Backward_Action, Output_Shape_Calculator > >:



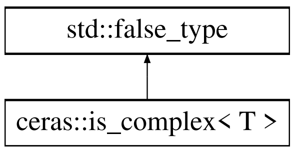
The documentation for this struct was generated from the following file:

- /home/feng/workspace/github.repo/ceras/include/operation.hpp

8.12 ceras::is_complex< T > Struct Template Reference

```
#include <complex_operator.hpp>
```

Inheritance diagram for ceras::is_complex< T >:



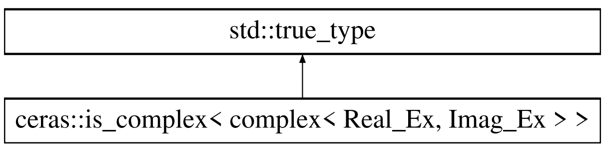
The documentation for this struct was generated from the following file:

- /home/feng/workspace/github.repo/ceras/include/complex_operator.hpp

8.13 ceras::is_complex< complex< Real_Ex, Imag_Ex > > Struct Template Reference

```
#include <complex_operator.hpp>
```

Inheritance diagram for ceras::is_complex< complex< Real_Ex, Imag_Ex > >:



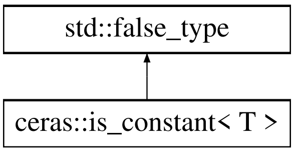
The documentation for this struct was generated from the following file:

- /home/feng/workspace/github.repo/ceras/include/complex_operator.hpp

8.14 ceras::is_constant< T > Struct Template Reference

```
#include <constant.hpp>
```

Inheritance diagram for ceras::is_constant< T >:



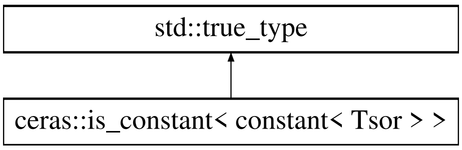
The documentation for this struct was generated from the following file:

- /home/feng/workspace/github.repo/ceras/include/constant.hpp

8.15 ceras::is_constant< constant< Tsor > > Struct Template Reference

```
#include <constant.hpp>
```

Inheritance diagram for ceras::is_constant< constant< Tsor > >:



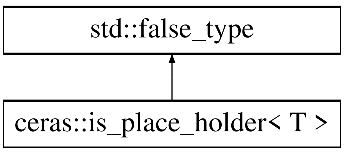
The documentation for this struct was generated from the following file:

- /home/feng/workspace/github.repo/ceras/include/constant.hpp

8.16 ceras::is_place_holder< T > Struct Template Reference

```
#include <place_holder.hpp>
```

Inheritance diagram for ceras::is_place_holder< T >:



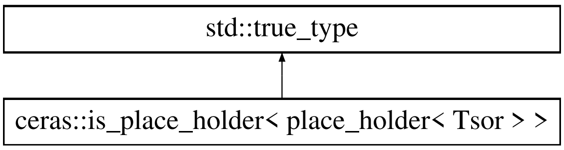
The documentation for this struct was generated from the following file:

- /home/feng/workspace/github.repo/ceras/include/place_holder.hpp

8.17 ceras::is_place_holder< place_holder< Tsor > > Struct Template Reference

```
#include <place_holder.hpp>
```

Inheritance diagram for ceras::is_place_holder< place_holder< Tsor > >:



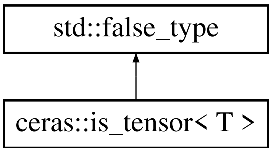
The documentation for this struct was generated from the following file:

- /home/feng/workspace/github.repo/ceras/include/place_holder.hpp

8.18 ceras::is_tensor< T > Struct Template Reference

```
#include <tensor.hpp>
```

Inheritance diagram for ceras::is_tensor< T >:



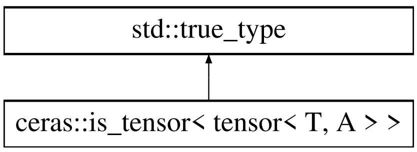
The documentation for this struct was generated from the following file:

- /home/feng/workspace/github.repo/ceras/include/tensor.hpp

8.19 ceras::is_tensor< tensor< T, A > > Struct Template Reference

```
#include <tensor.hpp>
```

Inheritance diagram for ceras::is_tensor< tensor< T, A > >:



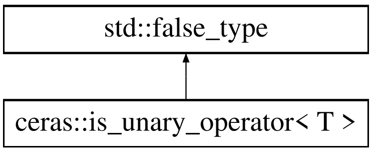
The documentation for this struct was generated from the following file:

- /home/feng/workspace/github.repo/ceras/include/tensor.hpp

8.20 ceras::is_unary_operator< T > Struct Template Reference

```
#include <operation.hpp>
```

Inheritance diagram for ceras::is_unary_operator< T >:



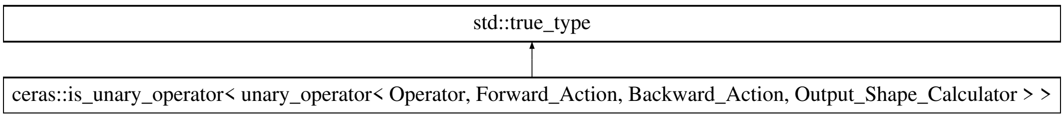
The documentation for this struct was generated from the following file:

- /home/feng/workspace/github.repo/ceras/include/operation.hpp

8.21 ceras::is_unary_operator< unary_operator< Operator, Forward_Action, Backward_Action, Output_Shape_Calculator > > Struct Template Reference

```
#include <operation.hpp>
```

Inheritance diagram for ceras::is_unary_operator< unary_operator< Operator, Forward_Action, Backward_Action, Output_Shape_Calculator > >:



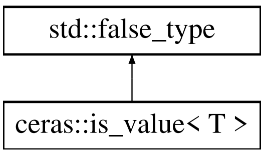
The documentation for this struct was generated from the following file:

- /home/feng/workspace/github.repo/ceras/include/operation.hpp

8.22 ceras::is_value< T > Struct Template Reference

```
#include <value.hpp>
```

Inheritance diagram for ceras::is_value< T >:



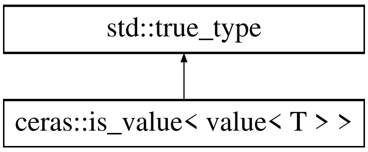
The documentation for this struct was generated from the following file:

- /home/feng/workspace/github.repo/ceras/include/value.hpp

8.23 ceras::is_value< value< T > > Struct Template Reference

```
#include <value.hpp>
```

Inheritance diagram for ceras::is_value< value< T > >:



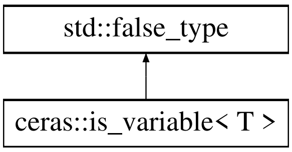
The documentation for this struct was generated from the following file:

- /home/feng/workspace/github.repo/ceras/include/value.hpp

8.24 ceras::is_variable< T > Struct Template Reference

```
#include <variable.hpp>
```

Inheritance diagram for ceras::is_variable< T >:



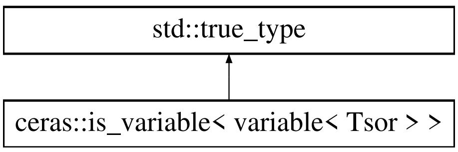
The documentation for this struct was generated from the following file:

- /home/feng/workspace/github.repo/ceras/include/variable.hpp

8.25 ceras::is_variable< variable< Tsor > > Struct Template Reference

```
#include <variable.hpp>
```

Inheritance diagram for ceras::is_variable< variable< Tsor > >:



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

- /home/feng/workspace/github.repo/ceras/include/variable.hpp

8.26 ceras::model< Ex, Ph > Struct Template Reference

```
#include <model.hpp>
```

Public Types

- typedef Ph input_layer_type
- typedef Ex output_layer_type

Public Member Functions

- input_layer_type input () const noexcept
- output_layer_type output () const noexcept
- model (input_layer_type const &place_holder, output_layer_type const &expression)
- template<Tensor Tsor>
  auto predict (Tsor const &input_tensor)
- template<Expression Exp>
  auto operator() (Exp const &ex) const noexcept
- template<typename Loss , typename Optimizer >
  auto compile (Loss const &l, Optimizer const &o)
- void trainable (bool t)
- void save_weights (std::string const &file)
- void load_weights (std::string const &file)
- void summary (std::string const &file_name=std::string{}) const noexcept
- constexpr model (Input_List const &input_layer, Output_List const &output_layer)
- constexpr auto input () const

  Returns the input layer(s) of the model in a 'list', which is are `place_holder`s.
- constexpr auto output () const

  Returns the output layer(s) of the model in a 'list', which is are expressions.
- template<Tensor Tsor>
  constexpr auto predict (Tsor const &input_tensor) const
- template<List Tsor_List>
  auto predict (Tsor_List const &input_tensor) const

Public Attributes

- output_layer_type expression_

  output layer of the model.
- input_layer_type place_holder_
- Input_List input_layer_
- Output_List output_layer_

8.26.1 Detailed Description

template<Expression Ex, Place_Holder Ph>
struct ceras::model< Ex, Ph >

Groups an input layer (a place holder) and an output layer (an expression template) into an object.

Template Parameters

Ex	The expression template for the output layer.
Ph	The place holder expression for the input layer

8.26.2 Member Typedef Documentation

8.26.2.1 input_layer_type

```
template<Expression Ex, Place_Holder Ph>
typedef Ph ceras::model< Ex, Ph >::input_layer_type
```

8.26.2.2 output_layer_type

```
template<Expression Ex, Place_Holder Ph>
typedef Ex ceras::model< Ex, Ph >::output_layer_type
```

8.26.3 Constructor & Destructor Documentation

8.26.3.1 model() [1/2]

```
template<Expression Ex, Place_Holder Ph>
ceras::model< Ex, Ph >::model (
             input_layer_type const & place_holder,
             output_layer_type const & expression )  [inline]
```

Parameters

place_holder	The input layer of the model, a place holder.
expression	The output layer of the model, a expression template.

Example code to generate a model:

```
auto input = Input();
auto l1 = relu( Dense( 1024, 28*28 )( input ) );
auto output = sigmoid( Dense( 10, 1024 )( l1 ) );
auto m = model{ input, output };
```

8.26.3.2 model() [2/2]

```
template<Expression Ex, Place_Holder Ph>
constexpr ceras::model< Ex, Ph >::model (
            Input_List const & input_layer,
            Output_List const & output_layer )  [inline], [constexpr]
```

8.26.4 Member Function Documentation

8.26.4.1 compile()

```
template<Expression Ex, Place_Holder Ph>
template<typename Loss , typename Optimizer >
auto ceras::model< Ex, Ph >::compile (
            Loss const & l,
            Optimizer const & o )  [inline]
```

Compile the model for training

Parameters

l	The loss to minimize.
o	The optimizer to do the optimization.

Returns

An instance of compiled_model.

Example useage:

```
model m{ ... };
unsigned long batch_size = 16;
float learning_rate = 0.001f;
auto cm = m.compile( MeanSquaredError(), SGD( batch_size, learning_rate ) );
```

8.26.4.2 input() [1/2]

```
template<Expression Ex, Place_Holder Ph>
constexpr auto ceras::model< Ex, Ph >::input ( ) const  [inline], [constexpr]
```

Returns the input layer(s) of the model in a 'list', which is are `place_holder`s.

8.26.4.3 input() [2/2]

```
template<Expression Ex, Place_Holder Ph>
input_layer_type ceras::model< Ex, Ph >::input ( ) const  [inline], [noexcept]
```

Returns the input layer of the model, which is a place_holder.

8.26.4.4 load_weights()

```
template<Expression Ex, Place_Holder Ph>
void ceras::model< Ex, Ph >::load_weights (
             std::string const & file )  [inline]
```

Loads all variables from a file

8.26.4.5 operator()()

```
template<Expression Ex, Place_Holder Ph>
template<Expression Exp>
auto ceras::model< Ex, Ph >::operator() (
             Exp const & ex ) const  [inline], [noexcept]
```

Generating a new expression by using the current model.

Parameters

ex	An expression that represents the input to the model.

Returns

An expression that replacing the input node with a new epxression.

Example code

```
auto x = Input(); // input, (28*28,)
auto y = Dense( 128, 28*28 )( x );
auto m1 = model( x, y ); // this model is [(28*28,) -> (128,)]
auto u = Input(); // new input, (32,)
auto v = Dense( 28*28, 32 )( u );
auto m2 = model( u, v );
auto input = Input(); // (32, )
auto ouptut = m1( m2( input ) ); // this new expression is [(32,) -> (28*28,) -> (128,)], note x is not in
      this expression any more
auto m = model( input, output ); // create a new model
```

8.26.4.6 output() [1/2]

```
template<Expression Ex, Place_Holder Ph>
constexpr auto ceras::model< Ex, Ph >::output ( ) const  [inline], [constexpr]
```

Returns the output layer(s) of the model in a 'list', which is are expressions.

8.26.4.7 output() [2/2]

```
template<Expression Ex, Place_Holder Ph>
output_layer_type ceras::model< Ex, Ph >::output ( ) const  [inline], [noexcept]
```

Returns the output layer of the model.

8.26.4.8 predict() [1/3]

```
template<Expression Ex, Place_Holder Ph>
template<Tensor Tsor>
auto ceras::model< Ex, Ph >::predict (
            Tsor const & input_tensor )  [inline]
```

Making prediction by binding the nput data to the `place_holder_` and evaluating `expression_`.

Parameters

input_tensor	The input samples.

Returns

The result this model predicts.

Example to predict

```
auto input = Input();
auto l1 = relu( Dense( 1024, 28*28 )( input ) );
auto output = sigmoid( Dense( 10, 1024 )( l1 ) );
// ...  train the model after defining a loss and an optimizer
auto m = model{ input, output };
auto test_data = random( {128, 28*28} ); // batch size is 128
auto result = model.predict( test_data ); // should produce an tensor of (128, 10)
```

8.26.4.9 predict() [2/3]

```
template<Expression Ex, Place_Holder Ph>
template<Tensor Tsor>
constexpr auto ceras::model< Ex, Ph >::predict (
            Tsor const & input_tensor ) const  [inline], [constexpr]
```

Making prediction by binding the nput data to the `place_holder_` and evaluating `expression_`.

Parameters

input_tensor	The input samples.

Returns

The result this model predicts.

Example to predict

```
auto input = Input();
auto l1 = relu( Dense( 1024, 28*28 )( input ) );
auto output = sigmoid( Dense( 10, 1024 )( l1 ) );
// ...  train the model after defining a loss and an optimizer
auto m = model{ input, output };
auto test_data = random( {128, 28*28} ); // batch size is 128
auto result = model.predict( test_data ); // should produce an tensor of (128, 10)
```

8.26.4.10 predict() [3/3]

```
template<Expression Ex, Place_Holder Ph>
template<List Tsor_List>
auto ceras::model< Ex, Ph >::predict (
            Tsor_List const & input_tensor ) const  [inline]
```

8.26.4.11 save_weights()

```
template<Expression Ex, Place_Holder Ph>
void ceras::model< Ex, Ph >::save_weights (
            std::string const & file )  [inline]
```

Writes all variables to a file

8.26.4.12 summary()

```
template<Expression Ex, Place_Holder Ph>
void ceras::model< Ex, Ph >::summary (
            std::string const & file_name = std::string{} ) const  [inline], [noexcept]
```

Print the model summary to console or to a file.

Parameters

file_name	The file to save the summary. If empty, the summary will be printed to console. Empty by default.

8.26.4.13 trainable()

```
template<Expression Ex, Place_Holder Ph>
void ceras::model< Ex, Ph >::trainable (
            bool t )  [inline]
```

8.26.5 Member Data Documentation

8.26.5.1 expression_

```
template<Expression Ex, Place_Holder Ph>
output_layer_type ceras::model< Ex, Ph >::expression_
```

output layer of the model.

8.26.5.2 input_layer_

```
template<Expression Ex, Place_Holder Ph>
Input_List ceras::model< Ex, Ph >::input_layer_
```

8.26.5.3 output_layer_

```
template<Expression Ex, Place_Holder Ph>
Output_List ceras::model< Ex, Ph >::output_layer_
```

8.26.5.4 place_holder_

```
template<Expression Ex, Place_Holder Ph>
input_layer_type ceras::model< Ex, Ph >::place_holder_
```

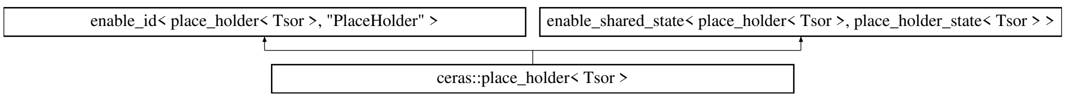
The documentation for this struct was generated from the following files:

- /home/feng/workspace/github.repo/ceras/include/model.hpp
- /home/feng/workspace/github.repo/ceras/include/xmodel.hpp

8.27 ceras::place_holder< Tsor > Struct Template Reference

```
#include <place_holder.hpp>
```

Inheritance diagram for ceras::place_holder< Tsor >:



Public Types

- typedef Tsor tensor_type

Public Member Functions

- place_holder (place_holder const &other)=default
- place_holder (place_holder &&other)=default
- place_holder & operator= (place_holder const &other)=default
- place_holder & operator= (place_holder &&other)=default
- place_holder ()
- place_holder (std::vector< unsigned long > const &shape_hint)
- void bind (Tsor data)
- Tsor const forward () const
- void reset () noexcept
- void backward (auto) const noexcept
- void shape (std::vector< unsigned long > const &shape_hint) noexcept
- std::vector< unsigned long > shape () const noexcept

8.27.1 Member Typedef Documentation

8.27.1.1 tensor_type

```
template<Tensor Tsor>
typedef Tsor ceras::place_holder< Tsor >::tensor_type
```

8.27.2 Constructor & Destructor Documentation

8.27.2.1 place_holder() [1/4]

```
template<Tensor Tsor>
ceras::place_holder< Tsor >::place_holder (
            place_holder< Tsor > const & other )  [default]
```

8.27.2.2 place_holder() [2/4]

```
template<Tensor Tsor>
ceras::place_holder< Tsor >::place_holder (
            place_holder< Tsor > && other )  [default]
```

8.27.2.3 place_holder() [3/4]

```
template<Tensor Tsor>
ceras::place_holder< Tsor >::place_holder ( )  [inline]
```

8.27.2.4 place_holder() [4/4]

```
template<Tensor Tsor>
ceras::place_holder< Tsor >::place_holder (
            std::vector< unsigned long > const & shape_hint )  [inline]
```

8.27.3 Member Function Documentation

8.27.3.1 backward()

```
template<Tensor Tsor>
void ceras::place_holder< Tsor >::backward (
            auto ) const [inline], [noexcept]
```

8.27.3.2 bind()

```
template<Tensor Tsor>
void ceras::place_holder< Tsor >::bind (
            Tsor data ) [inline]
```

8.27.3.3 forward()

```
template<Tensor Tsor>
Tsor const ceras::place_holder< Tsor >::forward ( ) const [inline]
```

8.27.3.4 operator=() [1/2]

```
template<Tensor Tsor>
place_holder & ceras::place_holder< Tsor >::operator= (
            place_holder< Tsor > && other ) [default]
```

8.27.3.5 operator=() [2/2]

```
template<Tensor Tsor>
place_holder & ceras::place_holder< Tsor >::operator= (
            place_holder< Tsor > const & other ) [default]
```

8.27.3.6 reset()

```
template<Tensor Tsor>
void ceras::place_holder< Tsor >::reset ( ) [inline], [noexcept]
```

8.27.3.7 shape() [1/2]

```
template<Tensor Tsor>
std::vector< unsigned long > ceras::place_holder< Tsor >::shape ( ) const  [inline], [noexcept]
```

8.27.3.8 shape() [2/2]

```
template<Tensor Tsor>
void ceras::place_holder< Tsor >::shape (
            std::vector< unsigned long > const & shape_hint )  [inline], [noexcept]
```

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

- /home/feng/workspace/github.repo/ceras/include/place_holder.hpp

8.28 ceras::place_holder_state< Tsor > Struct Template Reference

```
#include <place_holder.hpp>
```

Public Attributes

- Tsor data_
- std::vector< unsigned long > shape_hint_

8.28.1 Member Data Documentation

8.28.1.1 data_

```
template<Tensor Tsor>
Tsor ceras::place_holder_state< Tsor >::data_
```

8.28.1.2 shape_hint_

```
template<Tensor Tsor>
std::vector< unsigned long> ceras::place_holder_state< Tsor >::shape_hint_
```

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

- /home/feng/workspace/github.repo/ceras/include/place_holder.hpp

8.29 ceras::regularizer< Float > Struct Template Reference

```
#include <variable.hpp>
```

Public Types

- typedef Float value_type

Public Member Functions

- constexpr regularizer (value_type l1=0.0, value_type l2=0.0, bool synchronized=false) noexcept

Public Attributes

- value_type l1_
- value_type l2_
- bool synchronized_

8.29.1 Member Typedef Documentation

8.29.1.1 value_type

```
template<typename Float >
typedef Float ceras::regularizer< Float >::value_type
```

8.29.2 Constructor & Destructor Documentation

8.29.2.1 regularizer()

```
template<typename Float >
constexpr ceras::regularizer< Float >::regularizer (
            value_type l1 = 0.0,
            value_type l2 = 0.0,
            bool synchronized = false )  [inline], [constexpr], [noexcept]
```

8.29.3 Member Data Documentation

8.29.3.1 l1_

```
template<typename Float >
value_type ceras::regularizer< Float >::l1_
```

8.29.3.2 l2_

```
template<typename Float >
value_type ceras::regularizer< Float >::l2_
```

8.29.3.3 synchronized_

```
template<typename Float >
bool ceras::regularizer< Float >::synchronized_
```

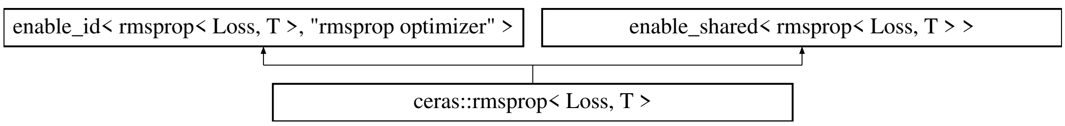
The documentation for this struct was generated from the following file:

- /home/feng/workspace/github.repo/ceras/include/variable.hpp

8.30 ceras::rmsprop< Loss, T > Struct Template Reference

```
#include <optimizer.hpp>
```

Inheritance diagram for ceras::rmsprop< Loss, T >:



Public Types

- typedef tensor< T > tensor_type

Public Member Functions

- rmsprop (Loss &loss, std::size_t batch_size, T learning_rate=1.0e-1, T rho=0.9, T decay=0.0) noexcept
- void forward ()

Public Attributes

- Loss & loss_
- T learning_rate_
- T rho_
- T decay_
- unsigned long iterations_

8.30.1 Member Typedef Documentation

8.30.1.1 tensor_type

```
template<typename Loss , typename T >
typedef tensor< T > ceras::rmsprop< Loss, T >::tensor_type
```

8.30.2 Constructor & Destructor Documentation

8.30.2.1 rmsprop()

```
template<typename Loss , typename T >
ceras::rmsprop< Loss, T >::rmsprop (
            Loss & loss,
            std::size_t batch_size,
            T learning_rate = 1.0e-1,
            T rho = 0.9,
            T decay = 0.0 )  [inline], [noexcept]
```

8.30.3 Member Function Documentation

8.30.3.1 forward()

```
template<typename Loss , typename T >
void ceras::rmsprop< Loss, T >::forward ( )  [inline]
```

8.30.4 Member Data Documentation

8.30.4.1 decay_

```
template<typename Loss , typename T >
T ceras::rmsprop< Loss, T >::decay_
```

8.30.4.2 iterations_

```
template<typename Loss , typename T >
unsigned long ceras::rmsprop< Loss, T >::iterations_
```

8.30.4.3 learning_rate_

```
template<typename Loss , typename T >
T ceras::rmsprop< Loss, T >::learning_rate_
```

8.30.4.4 loss_

```
template<typename Loss , typename T >
Loss& ceras::rmsprop< Loss, T >::loss_
```

8.30.4.5 rho_

```
template<typename Loss , typename T >
T ceras::rmsprop< Loss, T >::rho_
```

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

- /home/feng/workspace/github.repo/ceras/include/optimizer.hpp

8.31 ceras::ceras_private::session< Tsor > Struct Template Reference

```
#include <session.hpp>
```

Public Types

- typedef place_holder< Tsor > place_holder_type
- typedef variable< Tsor > variable_type
- typedef variable_state< Tsor > variable_state_type

Public Member Functions

- session ()
- session (session const &)=delete
- session (session &&)=default
- session & operator= (session const &)=delete
- session & operator= (session &&)=default
- session & rebind (place_holder_type &p_holder, Tsor const &value)
- session & bind (place_holder_type &p_holder, Tsor const &value)
- session & remember (variable_type const &v)
- template< typename Operation >
  auto run (Operation &op) const
- template< typename Operation >
  void tap (Operation &op) const
- void deserialize (std::string const &file_path)
- void serialize (std::string const &file_path) const
- void save (std::string const &file_path) const
- void restore (std::string const &file_path)
- void save_original (std::string const &file_path) const
- void restore_original (std::string const &file_path)
- ∼session ()

Public Attributes

- std::vector< place_holder_type > place_holders_
- std::map< int, variable_type > variables_

8.31.1 Member Typedef Documentation

8.31.1.1 place_holder_type

```
template<Tensor Tsor>
typedef place_holder<Tsor> ceras::ceras_private::session< Tsor >::place_holder_type
```

8.31.1.2 variable_state_type

```
template<Tensor Tsor>
typedef variable_state<Tsor> ceras::ceras_private::session< Tsor >::variable_state_type
```

8.31.1.3 variable_type

```
template<Tensor Tsor>
typedef variable<Tsor> ceras::ceras_private::session< Tsor >::variable_type
```

8.31.2 Constructor & Destructor Documentation

8.31.2.1 session() [1/3]

```
template<Tensor Tsor>
ceras::ceras_private::session< Tsor >::session ( ) [inline]
```

8.31.2.2 session() [2/3]

```
template<Tensor Tsor>
ceras::ceras_private::session< Tsor >::session (
            session< Tsor > const & ) [delete]
```

8.31.2.3 session() [3/3]

```
template<Tensor Tsor>
ceras::ceras_private::session< Tsor >::session (
            session< Tsor > && ) [default]
```

8.31.2.4 ∼session()

```
template<Tensor Tsor>
ceras::ceras_private::session< Tsor >::~session ( ) [inline]
```

8.31.3 Member Function Documentation

8.31.3.1 bind()

```
template<Tensor Tsor>
session & ceras::ceras_private::session< Tsor >::bind (
            place_holder_type & p_holder,
            Tsor const & value ) [inline]
```

8.31.3.2 deserialize()

```
template<Tensor Tsor>
void ceras::ceras_private::session< Tsor >::deserialize (
            std::string const & file_path )   [inline]
```

8.31.3.3 operator=() [1/2]

```
template<Tensor Tsor>
session & ceras::ceras_private::session< Tsor >::operator= (
            session< Tsor > &&  )   [default]
```

8.31.3.4 operator=() [2/2]

```
template<Tensor Tsor>
session & ceras::ceras_private::session< Tsor >::operator= (
            session< Tsor > const &  )   [delete]
```

8.31.3.5 rebind()

```
template<Tensor Tsor>
session & ceras::ceras_private::session< Tsor >::rebind (
            place_holder_type & p_holder,
            Tsor const & value )   [inline]
```

8.31.3.6 remember()

```
template<Tensor Tsor>
session & ceras::ceras_private::session< Tsor >::remember (
            variable_type const & v )   [inline]
```

8.31.3.7 restore()

```
template<Tensor Tsor>
void ceras::ceras_private::session< Tsor >::restore (
            std::string const & file_path )   [inline]
```

8.31.3.8 restore_original()

```
template<Tensor Tsor>
void ceras::ceras_private::session< Tsor >::restore_original (
            std::string const & file_path )  [inline]
```

8.31.3.9 run()

```
template<Tensor Tsor>
template<typename Operation >
auto ceras::ceras_private::session< Tsor >::run (
            Operation & op ) const  [inline]
```

8.31.3.10 save()

```
template<Tensor Tsor>
void ceras::ceras_private::session< Tsor >::save (
            std::string const & file_path ) const  [inline]
```

8.31.3.11 save_original()

```
template<Tensor Tsor>
void ceras::ceras_private::session< Tsor >::save_original (
            std::string const & file_path ) const  [inline]
```

8.31.3.12 serialize()

```
template<Tensor Tsor>
void ceras::ceras_private::session< Tsor >::serialize (
            std::string const & file_path ) const  [inline]
```

8.31.3.13 tap()

```
template<Tensor Tsor>
template<typename Operation >
void ceras::ceras_private::session< Tsor >::tap (
            Operation & op ) const  [inline]
```

8.31.4 Member Data Documentation

8.31.4.1 place_holders_

```
template<Tensor Tsor>
std::vector<place_holder_type> ceras::ceras_private::session< Tsor >::place_holders_
```

8.31.4.2 variables_

```
template<Tensor Tsor>
std::map<int, variable_type> ceras::ceras_private::session< Tsor >::variables_
```

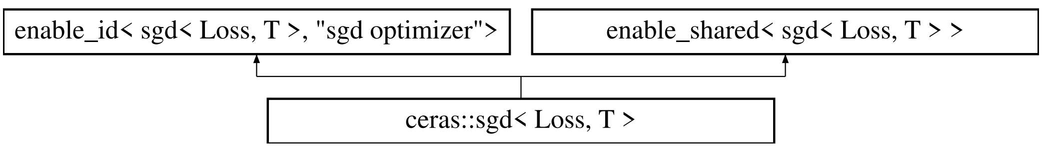
The documentation for this struct was generated from the following file:

- /home/feng/workspace/github.repo/ceras/include/session.hpp

8.32 ceras::sgd< Loss, T > Struct Template Reference

```
#include <optimizer.hpp>
```

Inheritance diagram for ceras::sgd< Loss, T >:



Public Types

- typedef tensor< T > tensor_type

Public Member Functions

- sgd (Loss &loss, std::size_t batch_size, T learning_rate=1.0e-1, T momentum=0.0, T decay=0.0, bool nesterov=false) noexcept
- void forward ()

Public Attributes

- Loss & loss_
- T learning_rate_
- T momentum_
- T decay_
- bool nesterov_
- unsigned long iterations_

8.32.1 Member Typedef Documentation

8.32.1.1 tensor_type

```
template<typename Loss , typename T >
typedef tensor< T > ceras::sgd< Loss, T >::tensor_type
```

8.32.2 Constructor & Destructor Documentation

8.32.2.1 sgd()

```
template<typename Loss , typename T >
ceras::sgd< Loss, T >::sgd (
             Loss & loss,
             std::size_t batch_size,
             T learning_rate = 1.0e-1,
             T momentum = 0.0,
             T decay = 0.0,
             bool nesterov = false )  [inline], [noexcept]
```

8.32.3 Member Function Documentation

8.32.3.1 forward()

```
template<typename Loss , typename T >
void ceras::sgd< Loss, T >::forward ( )  [inline]
```

8.32.4 Member Data Documentation

8.32.4.1 decay_

```
template<typename Loss , typename T >
T ceras::sgd< Loss, T >::decay_
```

8.32.4.2 iterations_

```
template<typename Loss , typename T >
unsigned long ceras::sgd< Loss, T >::iterations_
```

8.32.4.3 learning_rate_

```
template<typename Loss , typename T >
T ceras::sgd< Loss, T >::learning_rate_
```

8.32.4.4 loss_

```
template<typename Loss , typename T >
Loss& ceras::sgd< Loss, T >::loss_
```

8.32.4.5 momentum_

```
template<typename Loss , typename T >
T ceras::sgd< Loss, T >::momentum_
```

8.32.4.6 nesterov_

```
template<typename Loss , typename T >
bool ceras::sgd< Loss, T >::nesterov_
```

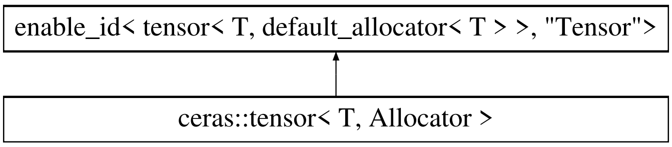
The documentation for this struct was generated from the following file:

- /home/feng/workspace/github.repo/ceras/include/optimizer.hpp

8.33 ceras::tensor< T, Allocator > Struct Template Reference

```
#include <tensor.hpp>
```

Inheritance diagram for ceras::tensor< T, Allocator >:



Public Types

- typedef T value_type
- typedef Allocator allocator
- typedef std::vector< T, Allocator > vector_type
- typedef std::shared_ptr< vector_type > shared_vector
- typedef tensor self_type

Public Member Functions

- tensor ()
- constexpr tensor (std::vector< unsigned long > const &shape, std::initializer_list< T > init, const Allocator &alloc=Allocator())

    Construct a vector with the specified shape, initialized value and a (default) allocator.
- constexpr tensor (std::vector< unsigned long > const &shape)

    Construct a vector with the specified shape. All values initialized to default. With a default constructed allocator.
- constexpr tensor (std::vector< unsigned long > const &shape, T init)

    Construct a vector with the specified shape and all values initialized to `init`*. With a default constructed allocator.*
- constexpr tensor (tensor const &other, unsigned long memory_offset) noexcept
- constexpr tensor (self_type const &other) noexcept

    Copy-ctor.
- constexpr tensor (self_type &&other) noexcept

    Move-ctor.
- constexpr self_type & operator= (self_type const &other) noexcept

    Copy-assignment.
- constexpr self_type & operator= (self_type &&other) noexcept

    Move-assignment.
- constexpr auto begin () noexcept

    Iterator to the first element of the tensor.
- constexpr auto begin () const noexcept

    Iterator to the first element of the tensor.
- constexpr auto cbegin () const noexcept

    Iterator to the first element of the tensor.
- constexpr auto end () noexcept

    Iterator to the element following the last element of the tensor.
- constexpr auto end () const noexcept

    Iterator to the element following the last element of the tensor.
- constexpr auto cend () const noexcept

Iterator to the element following the last element of the tensor.
- constexpr auto rbegin () noexcept

  Reverse iterator to the first element of the tensor.
- constexpr auto rbegin () const noexcept

  Reverse iterator to the first element of the tensor.
- constexpr auto crbegin () const noexcept

  Reverse iterator to the first element of the tensor.
- constexpr auto rend () noexcept

  Reverse iterator to the element following the last element of the tensor.
- constexpr auto rend () const noexcept

  Reverse iterator to the element following the last element of the tensor.
- constexpr auto crend () const noexcept

  Reverse iterator to the element following the last element of the tensor.
- constexpr auto front ()
- constexpr auto front () const
- constexpr auto back ()
- constexpr auto back () const
- constexpr unsigned long size () const noexcept

  Number of elements in the tensor.
- constexpr bool empty () const noexcept

  Check if the tensor has elements.
- constexpr self_type & reset (T val=T{0})
- constexpr unsigned long ndim () const noexcept

  Dimension of the tensor.
- constexpr std::vector< unsigned long > const & shape () const noexcept

  Shape of the tensor.
- constexpr self_type & deep_copy (self_type const &other)

  A deep copy of the tensor.
- constexpr self_type const deep_copy () const
- constexpr self_type const copy () const
- constexpr value_type & operator[] (unsigned long idx)
- constexpr value_type const & operator[] (unsigned long idx) const
- constexpr self_type & resize (std::vector< unsigned long > const &new_shape)

  Resize the tensor with a new shape.
- constexpr self_type & reshape (std::vector< unsigned long > const &new_shape)

  Reshape tensor. -1 indicates the dimension needs recalculating.
- constexpr self_type & shrink_to (std::vector< unsigned long > const &new_shape)
- constexpr self_type & creep_to (unsigned long new_memory_offset)
- constexpr value_type * data () noexcept

  Returns pointer to the underlying array serving as element storage.
- constexpr const value_type * data () const noexcept

  Returns pointer to the underlying array serving as element storage.
- template<typename Function >
  constexpr self_type & map (Function const &f)

  Applying element-wise operation on each element in the tensor.
- constexpr self_type & operator+= (self_type const &other)
- constexpr self_type & operator+= (value_type x)
- constexpr self_type & operator-= (self_type const &other)
- constexpr self_type & operator-= (value_type x)
- constexpr self_type & operator*= (self_type const &other)
- constexpr self_type & operator*= (value_type x)
- constexpr self_type & operator/= (self_type const &other)

- constexpr self_type & operator/= (value_type x)
- constexpr self_type const operator- () const
- constexpr value_type as_scalar () const noexcept
- template< typename U >
  constexpr auto as_type () const noexcept
- tensor slice (unsigned long m, unsigned long n) const noexcept

Public Attributes

- std::vector< unsigned long > shape_
- unsigned long memory_offset_
- shared_vector vector_

8.33.1 Member Typedef Documentation

8.33.1.1 allocator

```
template<typename T , typename Allocator = default_allocator<T>>
typedef Allocator ceras::tensor< T, Allocator >::allocator
```

8.33.1.2 self_type

```
template<typename T , typename Allocator = default_allocator<T>>
typedef tensor ceras::tensor< T, Allocator >::self_type
```

8.33.1.3 shared_vector

```
template<typename T , typename Allocator = default_allocator<T>>
typedef std::shared_ptr<vector_type> ceras::tensor< T, Allocator >::shared_vector
```

8.33.1.4 value_type

```
template<typename T , typename Allocator = default_allocator<T>>
typedef T ceras::tensor< T, Allocator >::value_type
```

8.33.1.5 vector_type

```
template<typename T , typename Allocator = default_allocator<T>>
typedef std::vector<T, Allocator> ceras::tensor< T, Allocator >::vector_type
```

8.33.2 Constructor & Destructor Documentation

8.33.2.1 tensor() [1/7]

```
template<typename T , typename Allocator = default_allocator<T>>
ceras::tensor< T, Allocator >::tensor ( )  [inline]
```

@breif Construct an empty vector

8.33.2.2 tensor() [2/7]

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr ceras::tensor< T, Allocator >::tensor (
            std::vector< unsigned long > const & shape,
            std::initializer_list< T > init,
            const Allocator & alloc = Allocator() )  [inline], [constexpr]
```

Construct a vector with the specified shape, initialized value and a (default) allocator.

8.33.2.3 tensor() [3/7]

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr ceras::tensor< T, Allocator >::tensor (
            std::vector< unsigned long > const & shape )  [inline], [constexpr]
```

Construct a vector with the specified shape. All values initialized to default. With a default constructed allocator.

8.33.2.4 tensor() [4/7]

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr ceras::tensor< T, Allocator >::tensor (
            std::vector< unsigned long > const & shape,
            T init )  [inline], [constexpr]
```

Construct a vector with the specified shape and all values initialized to `init`. With a default constructed allocator.

8.33.2.5 tensor() [5/7]

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr ceras::tensor< T, Allocator >::tensor (
            tensor< T, Allocator > const & other,
            unsigned long memory_offset )  [inline], [constexpr], [noexcept]
```

8.33.2.6 tensor() [6/7]

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr ceras::tensor< T, Allocator >::tensor (
            self_type const & other )  [inline], [constexpr], [noexcept]
```

Copy-ctor.

8.33.2.7 tensor() [7/7]

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr ceras::tensor< T, Allocator >::tensor (
            self_type && other )  [inline], [constexpr], [noexcept]
```

Move-ctor.

8.33.3 Member Function Documentation

8.33.3.1 as_scalar()

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr value_type ceras::tensor< T, Allocator >::as_scalar ( ) const  [inline], [constexpr],
[noexcept]
```

8.33.3.2 as_type()

```
template<typename T , typename Allocator = default_allocator<T>>
template<typename U >
constexpr auto ceras::tensor< T, Allocator >::as_type ( ) const  [inline], [constexpr], [noexcept]
```

8.33.3.3 back() [1/2]

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr auto ceras::tensor< T, Allocator >::back ( )  [inline], [constexpr]
```

@breif Reference to the last element in the tensor.

8.33.3.4 back() [2/2]

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr auto ceras::tensor< T, Allocator >::back ( ) const  [inline], [constexpr]
```

@breif Reference to the last element in the tensor.

8.33.3.5 begin() [1/2]

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr auto ceras::tensor< T, Allocator >::begin ( ) const  [inline], [constexpr], [noexcept]
```

Iterator to the first element of the tensor.

8.33.3.6 begin() [2/2]

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr auto ceras::tensor< T, Allocator >::begin ( )  [inline], [constexpr], [noexcept]
```

Iterator to the first element of the tensor.

8.33.3.7 cbegin()

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr auto ceras::tensor< T, Allocator >::cbegin ( ) const  [inline], [constexpr], [noexcept]
```

Iterator to the first element of the tensor.

8.33.3.8 cend()

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr auto ceras::tensor< T, Allocator >::cend ( ) const  [inline], [constexpr], [noexcept]
```

Iterator to the element following the last element of the tensor.

8.33.3.9 copy()

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr self_type const ceras::tensor< T, Allocator >::copy ( ) const  [inline], [constexpr]
```

8.33.3.10 crbegin()

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr auto ceras::tensor< T, Allocator >::crbegin ( ) const  [inline], [constexpr], [noexcept]
```

Reverse iterator to the first element of the tensor.

8.33.3.11 creep_to()

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr self_type & ceras::tensor< T, Allocator >::creep_to (
            unsigned long new_memory_offset )  [inline], [constexpr]
```

8.33.3.12 crend()

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr auto ceras::tensor< T, Allocator >::crend ( ) const  [inline], [constexpr], [noexcept]
```

Reverse iterator to the element following the last element of the tensor.

8.33.3.13 data() [1/2]

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr const value_type * ceras::tensor< T, Allocator >::data ( ) const  [inline], [constexpr],
[noexcept]
```

Returns pointer to the underlying array serving as element storage.

The pointer is such that range [data(); data() + size()) is always a valid range, even if the container is empty (data() is not dereferenceable in that case).

8.33.3.14 data() [2/2]

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr value_type * ceras::tensor< T, Allocator >::data ( )  [inline], [constexpr], [noexcept]
```

Returns pointer to the underlying array serving as element storage.

The pointer is such that range [data(); data() + size()) is always a valid range, even if the container is empty (data() is not dereferenceable in that case).

8.33.3.15 deep_copy() [1/2]

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr self_type const ceras::tensor< T, Allocator >::deep_copy ( ) const  [inline], [constexpr]
```

8.33.3.16 deep_copy() [2/2]

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr self_type & ceras::tensor< T, Allocator >::deep_copy (
             self_type const & other )  [inline], [constexpr]
```

A deep copy of the tensor.

8.33.3.17 empty()

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr bool ceras::tensor< T, Allocator >::empty ( ) const  [inline], [constexpr], [noexcept]
```

Check if the tensor has elements.

8.33.3.18 end() [1/2]

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr auto ceras::tensor< T, Allocator >::end ( ) const  [inline], [constexpr], [noexcept]
```

Iterator to the element following the last element of the tensor.

8.33.3.19 end() [2/2]

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr auto ceras::tensor< T, Allocator >::end ( )  [inline], [constexpr], [noexcept]
```

Iterator to the element following the last element of the tensor.

8.33.3.20 front() [1/2]

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr auto ceras::tensor< T, Allocator >::front ( )  [inline], [constexpr]
```

@breif Reference to the first element in the tensor.

8.33.3.21 front() [2/2]

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr auto ceras::tensor< T, Allocator >::front ( ) const  [inline], [constexpr]
```

@breif Reference to the first element in the tensor.

8.33.3.22 map()

```
template<typename T , typename Allocator = default_allocator<T>>
template<typename Function >
constexpr self_type & ceras::tensor< T, Allocator >::map (
             Function const & f )  [inline], [constexpr]
```

Applying element-wise operation on each element in the tensor.

```
tensor<double> x{...};
x.map( []( double v ){ return 1.0/v+1.0; } );
```

8.33.3.23 ndim()

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr unsigned long ceras::tensor< T, Allocator >::ndim ( ) const  [inline], [constexpr],
[noexcept]
```

Dimension of the tensor.

8.33.3.24 operator*=() [1/2]

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr self_type & ceras::tensor< T, Allocator >::operator*= (
             self_type const & other )  [inline], [constexpr]
```

8.33.3.25 operator*=() [2/2]

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr self_type & ceras::tensor< T, Allocator >::operator*= (
             value_type x )  [inline], [constexpr]
```

8.33.3.26 operator+=() [1/2]

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr self_type & ceras::tensor< T, Allocator >::operator+= (
             self_type const & other )  [inline], [constexpr]
```

8.33.3.27 operator+=() [2/2]

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr self_type & ceras::tensor< T, Allocator >::operator+= (
            value_type x )  [inline], [constexpr]
```

8.33.3.28 operator-()

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr self_type const ceras::tensor< T, Allocator >::operator- ( ) const  [inline], [constexpr]
```

8.33.3.29 operator-=() [1/2]

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr self_type & ceras::tensor< T, Allocator >::operator-= (
            self_type const & other )  [inline], [constexpr]
```

8.33.3.30 operator-=() [2/2]

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr self_type & ceras::tensor< T, Allocator >::operator-= (
            value_type x )  [inline], [constexpr]
```

8.33.3.31 operator/=() [1/2]

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr self_type & ceras::tensor< T, Allocator >::operator/= (
            self_type const & other )  [inline], [constexpr]
```

8.33.3.32 operator/=() [2/2]

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr self_type & ceras::tensor< T, Allocator >::operator/= (
            value_type x )  [inline], [constexpr]
```

8.33.3.33 operator=() [1/2]

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr self_type & ceras::tensor< T, Allocator >::operator= (
            self_type && other )  [inline], [constexpr], [noexcept]
```

Move-assignment.

8.33.3.34 operator=() [2/2]

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr self_type & ceras::tensor< T, Allocator >::operator= (
            self_type const & other )  [inline], [constexpr], [noexcept]
```

Copy-assignment.

8.33.3.35 operator[]() [1/2]

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr value_type & ceras::tensor< T, Allocator >::operator[] (
            unsigned long idx )  [inline], [constexpr]
```

8.33.3.36 operator[]() [2/2]

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr value_type const & ceras::tensor< T, Allocator >::operator[] (
            unsigned long idx ) const  [inline], [constexpr]
```

8.33.3.37 rbegin() [1/2]

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr auto ceras::tensor< T, Allocator >::rbegin ( ) const  [inline], [constexpr], [noexcept]
```

Reverse iterator to the first element of the tensor.

8.33.3.38 rbegin() [2/2]

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr auto ceras::tensor< T, Allocator >::rbegin ( )  [inline], [constexpr], [noexcept]
```

Reverse iterator to the first element of the tensor.

8.33.3.39 rend() [1/2]

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr auto ceras::tensor< T, Allocator >::rend ( ) const  [inline], [constexpr], [noexcept]
```

Reverse iterator to the element following the last element of the tensor.

8.33.3.40 rend() [2/2]

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr auto ceras::tensor< T, Allocator >::rend ( )  [inline], [constexpr], [noexcept]
```

Reverse iterator to the element following the last element of the tensor.

8.33.3.41 reset()

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr self_type & ceras::tensor< T, Allocator >::reset (
            T val = T{0} )  [inline], [constexpr]
```

Resetting all elements in the tensor to a fixed value (default to 0), without change the shape.

Example code:

```
tensor<float> ts;
ts.reset( 0.0f );
```

8.33.3.42 reshape()

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr self_type & ceras::tensor< T, Allocator >::reshape (
            std::vector< unsigned long > const & new_shape )  [inline], [constexpr]
```

Reshape tensor. -1 indicates the dimension needs recalculating.

```
tensor<float> t{ {2, 3, 4} };
auto t1 = t.reshape( {3, 8} );
auto t2 = t.reshape( {1, 4, -1UL} );
```

8.33.3.43 resize()

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr self_type & ceras::tensor< T, Allocator >::resize (
            std::vector< unsigned long > const & new_shape )  [inline], [constexpr]
```

Resize the tensor with a new shape.

8.33.3.44 shape()

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr std::vector< unsigned long > const & ceras::tensor< T, Allocator >::shape ( ) const
[inline], [constexpr], [noexcept]
```

Shape of the tensor.

8.33.3.45 shrink_to()

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr self_type & ceras::tensor< T, Allocator >::shrink_to (
            std::vector< unsigned long > const & new_shape )  [inline], [constexpr]
```

8.33.3.46 size()

```
template<typename T , typename Allocator = default_allocator<T>>
constexpr unsigned long ceras::tensor< T, Allocator >::size ( ) const  [inline], [constexpr],
[noexcept]
```

Number of elements in the tensor.

8.33.3.47 slice()

```
template<typename T , typename Allocator = default_allocator<T>>
tensor ceras::tensor< T, Allocator >::slice (
            unsigned long m,
            unsigned long n ) const  [inline], [noexcept]
```

8.33.4 Member Data Documentation

8.33.4.1 memory_offset_

```
template<typename T , typename Allocator = default_allocator<T>>
unsigned long ceras::tensor< T, Allocator >::memory_offset_
```

8.33.4.2 shape_

```
template<typename T , typename Allocator = default_allocator<T>>
std::vector<unsigned long> ceras::tensor< T, Allocator >::shape_
```

8.33.4.3 vector_

```
template<typename T , typename Allocator = default_allocator<T>>
shared_vector ceras::tensor< T, Allocator >::vector_
```

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

- /home/feng/workspace/github.repo/ceras/include/tensor.hpp

8.34 ceras::tensor_deduction< L, R > Struct Template Reference

```
#include <value.hpp>
```

Public Types

- using op_type = std::conditional< is_value_v< L >, R, L >::type
- using tensor_type = std::remove_cv_t< decltype(std::declval< op_type >().forward())>

8.34.1 Member Typedef Documentation

8.34.1.1 op_type

```
template<typename L , typename R >
using ceras::tensor_deduction< L, R >::op_type = std::conditional<is_value_v<L>, R, L>::type
```

8.34.1.2 tensor_type

```
template<typename L , typename R >
using ceras::tensor_deduction< L, R >::tensor_type = std::remove_cv_t<decltype(std::declval<op_type>().forwa
```

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

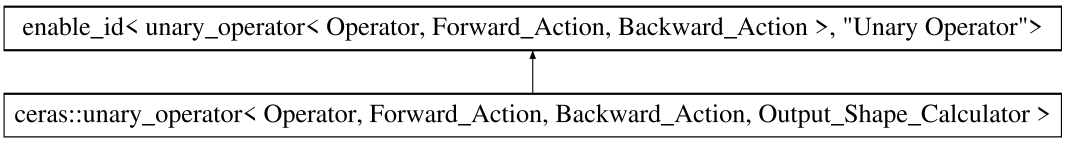
- /home/feng/workspace/github.repo/ceras/include/value.hpp

8.35 ceras::unary_operator< Operator, Forward_Action, Backward_Action, Output_Shape_Calculator > Struct Template Reference

A unary operator is composed of a.) an input expression, b.) a forward action and c.) a backward action.

```
#include <operation.hpp>
```

Inheritance diagram for ceras::unary_operator< Operator, Forward_Action, Backward_Action, Output_Shape_Calculator >:



Public Types

- typedef decltype(std::declval< Forward_Action >()(std::declval< decltype(op_)>().forward())) tensor_type

Public Member Functions

- unary_operator (Operator const &op, Forward_Action const &forward_action, Backward_Action const &backward_action, Output_Shape_Calculator const &output_shape_calculator) noexcept
- auto forward ()
- void backward (tensor_type const &grad)
- std::vector< unsigned long > shape () const noexcept

  Calculate the output tensor shape.

Public Attributes

- Operator op_
- Forward_Action forward_action_
- Backward_Action backward_action_
- Output_Shape_Calculator output_shape_calculator_
- tensor_type input_data_
- tensor_type output_data_

8.35.1 Detailed Description

template<typename Operator, typename Forward_Action, typename Backward_Action, typename Output_Shape_Calculator = identity_output_shape_calculator>
struct ceras::unary_operator< Operator, Forward_Action, Backward_Action, Output_Shape_Calculator >

A unary operator is composed of a.) an input expression, b.) a forward action and c.) a backward action.

8.35.2 Member Typedef Documentation

8.35.2.1 tensor_type

```
template<typename Operator , typename Forward_Action , typename Backward_Action , typename
Output_Shape_Calculator = identity_output_shape_calculator>
typedef decltype( std::declval<Forward_Action>()( std::declval<decltype(op_)>().forward() ) )
ceras::unary_operator< Operator, Forward_Action, Backward_Action, Output_Shape_Calculator >↩
::tensor_type
```

8.35.3 Constructor & Destructor Documentation

8.35.3.1 unary_operator()

```
template<typename Operator , typename Forward_Action , typename Backward_Action , typename
Output_Shape_Calculator = identity_output_shape_calculator>
ceras::unary_operator< Operator, Forward_Action, Backward_Action, Output_Shape_Calculator >↩
::unary_operator (
            Operator const & op,
            Forward_Action const & forward_action,
            Backward_Action const & backward_action,
            Output_Shape_Calculator const & output_shape_calculator )  [inline], [noexcept]
```

8.35.4 Member Function Documentation

8.35.4.1 backward()

```
template<typename Operator , typename Forward_Action , typename Backward_Action , typename
Output_Shape_Calculator = identity_output_shape_calculator>
void ceras::unary_operator< Operator, Forward_Action, Backward_Action, Output_Shape_Calculator
>::backward (
            tensor_type const & grad )  [inline]
```

8.35.4.2 forward()

```
template<typename Operator , typename Forward_Action , typename Backward_Action , typename
Output_Shape_Calculator = identity_output_shape_calculator>
auto ceras::unary_operator< Operator, Forward_Action, Backward_Action, Output_Shape_Calculator
>::forward ( )  [inline]
```

8.35.4.3 shape()

```
template<typename Operator , typename Forward_Action , typename Backward_Action , typename
Output_Shape_Calculator = identity_output_shape_calculator>
std::vector< unsigned long > ceras::unary_operator< Operator, Forward_Action, Backward_←
Action, Output_Shape_Calculator >::shape ( ) const  [inline], [noexcept]
```

Calculate the output tensor shape.

8.35.5 Member Data Documentation

8.35.5.1 backward_action_

```
template<typename Operator , typename Forward_Action , typename Backward_Action , typename
Output_Shape_Calculator = identity_output_shape_calculator>
Backward_Action ceras::unary_operator< Operator, Forward_Action, Backward_Action, Output_←
Shape_Calculator >::backward_action_
```

8.35.5.2 forward_action_

```
template<typename Operator , typename Forward_Action , typename Backward_Action , typename
Output_Shape_Calculator = identity_output_shape_calculator>
Forward_Action ceras::unary_operator< Operator, Forward_Action, Backward_Action, Output_←
Shape_Calculator >::forward_action_
```

8.35.5.3 input_data_

```
template<typename Operator , typename Forward_Action , typename Backward_Action , typename
Output_Shape_Calculator = identity_output_shape_calculator>
tensor_type ceras::unary_operator< Operator, Forward_Action, Backward_Action, Output_Shape_←
Calculator >::input_data_
```

8.35.5.4 op_

```
template<typename Operator , typename Forward_Action , typename Backward_Action , typename
Output_Shape_Calculator = identity_output_shape_calculator>
Operator ceras::unary_operator< Operator, Forward_Action, Backward_Action, Output_Shape_←
Calculator >::op_
```

8.35.5.5 output_data_

```
template<typename Operator , typename Forward_Action , typename Backward_Action , typename
Output_Shape_Calculator = identity_output_shape_calculator>
tensor_type ceras::unary_operator< Operator, Forward_Action, Backward_Action, Output_Shape_↩
Calculator >::output_data_
```

8.35.5.6 output_shape_calculator_

```
template<typename Operator , typename Forward_Action , typename Backward_Action , typename
Output_Shape_Calculator = identity_output_shape_calculator>
Output_Shape_Calculator ceras::unary_operator< Operator, Forward_Action, Backward_Action,
Output_Shape_Calculator >::output_shape_calculator_
```

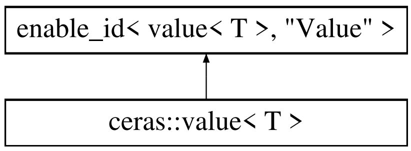
The documentation for this struct was generated from the following file:

- /home/feng/workspace/github.repo/ceras/include/operation.hpp

8.36 ceras::value< T > Struct Template Reference

```
#include <value.hpp>
```

Inheritance diagram for ceras::value< T >:



Public Types

- typedef T value_type
- typedef tensor< value_type > tensor_type

Public Member Functions

- value ()=delete
- value (value_type v) noexcept
- value (value const &) noexcept=default
- value (value &&) noexcept=default
- value & operator= (value const &) noexcept=default
- value & operator= (value &&) noexcept=default
- void backward (auto) noexcept
- template<Tensor Tsor>
  Tsor const forward (Tsor const &refer) const
- std::vector< unsigned long > shape () const noexcept

Public Attributes

- value_type data_

8.36.1 Member Typedef Documentation

8.36.1.1 tensor_type

```
template<typename T >
typedef tensor<value_type> ceras::value< T >::tensor_type
```

8.36.1.2 value_type

```
template<typename T >
typedef T ceras::value< T >::value_type
```

8.36.2 Constructor & Destructor Documentation

8.36.2.1 value() [1/4]

```
template<typename T >
ceras::value< T >::value ( )  [delete]
```

8.36.2.2 value() [2/4]

```
template<typename T >
ceras::value< T >::value (
            value_type v )  [inline], [noexcept]
```

8.36.2.3 value() [3/4]

```
template<typename T >
ceras::value< T >::value (
            value< T > const & )  [default], [noexcept]
```

8.36.2.4 value() [4/4]

```
template<typename T >
ceras::value< T >::value (
            value< T > && ) [default], [noexcept]
```

8.36.3 Member Function Documentation

8.36.3.1 backward()

```
template<typename T >
void ceras::value< T >::backward (
            auto ) [inline], [noexcept]
```

8.36.3.2 forward()

```
template<typename T >
template<Tensor Tsor>
Tsor const ceras::value< T >::forward (
            Tsor const & refer ) const [inline]
```

8.36.3.3 operator=() [1/2]

```
template<typename T >
value & ceras::value< T >::operator= (
            value< T > && ) [default], [noexcept]
```

8.36.3.4 operator=() [2/2]

```
template<typename T >
value & ceras::value< T >::operator= (
            value< T > const & ) [default], [noexcept]
```

8.36.3.5 shape()

```
template<typename T >
std::vector< unsigned long > ceras::value< T >::shape ( ) const [inline], [noexcept]
```

8.36.4 Member Data Documentation

8.36.4.1 data_

```
template<typename T >
value_type ceras::value< T >::data_
```

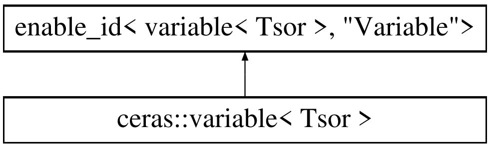
The documentation for this struct was generated from the following file:

- /home/feng/workspace/github.repo/ceras/include/value.hpp

8.37 ceras::variable< Tsor > Struct Template Reference

```
#include <variable.hpp>
```

Inheritance diagram for ceras::variable< Tsor >:



Public Types

- typedef Tsor tensor_type
- typedef tensor_type::value_type value_type

Public Member Functions

- variable (tensor_type const &data, value_type l1=value_type{0}, value_type l2=value_type{0}, bool trainable=true)
- variable () noexcept
- variable (variable const &other)=default
- variable (variable &&)=default
- variable & operator= (variable &&)=default
- variable & operator= (variable const &other)=default
- tensor_type const forward () noexcept
- void backward (auto const &grad) noexcept
- std::vector< std::size_t > shape () const noexcept
- std::vector< tensor_type > & contexts ()
- std::vector< tensor_type > contexts () const
- tensor_type & data ()
- tensor_type data () const
- tensor_type & gradient ()
- tensor_type gradient () const
- void reset ()
- bool trainable () const noexcept
- void trainable (bool t)

Public Attributes

- std::shared_ptr< variable_state< tensor_type > > state_
- regularizer< value_type > regularizer_
- bool trainable_

8.37.1 Member Typedef Documentation

8.37.1.1 tensor_type

```
template<Tensor Tsor>
typedef Tsor ceras::variable< Tsor >::tensor_type
```

8.37.1.2 value_type

```
template<Tensor Tsor>
typedef tensor_type::value_type ceras::variable< Tsor >::value_type
```

8.37.2 Constructor & Destructor Documentation

8.37.2.1 variable() [1/4]

```
template<Tensor Tsor>
ceras::variable< Tsor >::variable (
            tensor_type const & data,
            value_type l1 = value_type{0},
            value_type l2 = value_type{0},
            bool trainable = true ) [inline]
```

8.37.2.2 variable() [2/4]

```
template<Tensor Tsor>
ceras::variable< Tsor >::variable ( ) [inline], [noexcept]
```

8.37.2.3 variable() [3/4]

```
template<Tensor Tsor>
ceras::variable< Tsor >::variable (
            variable< Tsor > const & other )  [default]
```

8.37.2.4 variable() [4/4]

```
template<Tensor Tsor>
ceras::variable< Tsor >::variable (
            variable< Tsor > &&  )  [default]
```

8.37.3 Member Function Documentation

8.37.3.1 backward()

```
template<Tensor Tsor>
void ceras::variable< Tsor >::backward (
            auto const & grad )  [inline], [noexcept]
```

8.37.3.2 contexts() [1/2]

```
template<Tensor Tsor>
std::vector< tensor_type > & ceras::variable< Tsor >::contexts ( )  [inline]
```

8.37.3.3 contexts() [2/2]

```
template<Tensor Tsor>
std::vector< tensor_type > ceras::variable< Tsor >::contexts ( ) const  [inline]
```

8.37.3.4 data() [1/2]

```
template<Tensor Tsor>
tensor_type & ceras::variable< Tsor >::data ( )  [inline]
```

8.37.3.5 data() [2/2]

```
template<Tensor Tsor>
tensor_type ceras::variable< Tsor >::data ( ) const  [inline]
```

8.37.3.6 forward()

```
template<Tensor Tsor>
tensor_type const ceras::variable< Tsor >::forward ( )  [inline], [noexcept]
```

8.37.3.7 gradient() [1/2]

```
template<Tensor Tsor>
tensor_type & ceras::variable< Tsor >::gradient ( )  [inline]
```

8.37.3.8 gradient() [2/2]

```
template<Tensor Tsor>
tensor_type ceras::variable< Tsor >::gradient ( ) const  [inline]
```

8.37.3.9 operator=() [1/2]

```
template<Tensor Tsor>
variable & ceras::variable< Tsor >::operator= (
            variable< Tsor > &&  )  [default]
```

8.37.3.10 operator=() [2/2]

```
template<Tensor Tsor>
variable & ceras::variable< Tsor >::operator= (
            variable< Tsor > const & other )  [default]
```

8.37.3.11 reset()

```
template<Tensor Tsor>
void ceras::variable< Tsor >::reset ( )  [inline]
```

8.37.3.12 shape()

```
template<Tensor Tsor>
std::vector< std::size_t > ceras::variable< Tsor >::shape ( ) const  [inline], [noexcept]
```

8.37.3.13 trainable() [1/2]

```
template<Tensor Tsor>
bool ceras::variable< Tsor >::trainable ( ) const  [inline], [noexcept]
```

8.37.3.14 trainable() [2/2]

```
template<Tensor Tsor>
void ceras::variable< Tsor >::trainable (
            bool t )  [inline]
```

8.37.4 Member Data Documentation

8.37.4.1 regularizer_

```
template<Tensor Tsor>
regularizer<value_type> ceras::variable< Tsor >::regularizer_
```

8.37.4.2 state_

```
template<Tensor Tsor>
std::shared_ptr<variable_state<tensor_type> > ceras::variable< Tsor >::state_
```

8.37.4.3 trainable_

```
template<Tensor Tsor>
bool ceras::variable< Tsor >::trainable_
```

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

- /home/feng/workspace/github.repo/ceras/include/variable.hpp

8.38 ceras::variable_state< Tsor > Struct Template Reference

```
#include <variable.hpp>
```

Public Attributes

- Tsor data_
- Tsor gradient_
- std::vector< Tsor > contexts_

8.38.1 Member Data Documentation

8.38.1.1 contexts_

```
template<Tensor Tsor>
std::vector<Tsor> ceras::variable_state< Tsor >::contexts_
```

8.38.1.2 data_

```
template<Tensor Tsor>
Tsor ceras::variable_state< Tsor >::data_
```

8.38.1.3 gradient_

```
template<Tensor Tsor>
Tsor ceras::variable_state< Tsor >::gradient_
```

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

- /home/feng/workspace/github.repo/ceras/include/variable.hpp

Chapter 9

File Documentation

9.1 /home/feng/workspace/github.repo/ceras/include/activation.hpp File Reference

```
#include "./operation.hpp"
#include "./tensor.hpp"
#include "./utils/range.hpp"
#include "./utils/better_assert.hpp"
#include "./utils/for_each.hpp"
#include "./utils/context_cast.hpp"
```

Namespaces

- namespace ceras

Functions

- template<std::floating_point Float>
  constexpr auto ceras::heaviside_step (Float f) noexcept

  Step activation function, an unary operator.
- template<Expression Ex>
  constexpr auto ceras::soft_sign (Ex const &ex) noexcept
- template<Expression Ex>
  constexpr auto ceras::unit_step (Ex const &ex) noexcept
- template<Expression Ex>
  constexpr auto ceras::binary_step (Ex const &ex) noexcept
- template<Expression Ex>
  constexpr auto ceras::gaussian (Ex const &ex) noexcept

  Gaussian activation function, an unary operator.
- template<Expression Ex>
  constexpr auto ceras::softmax (Ex const &ex) noexcept

  Softmax activation function, an unary operator.
- template<Expression Ex>
  auto ceras::selu (Ex const &ex) noexcept

Scaled Exponential Linear Unit (SELU) activation function, an unary operator. If `x>0`*, returns 1.0507 x; Otherwise, returns 1.67326*1.0507*(exp(x)-1)*

- template<Expression Ex>
  auto ceras::softplus (Ex const &ex) noexcept

  Softplus function, an unary operator. Returns `log(exp(x)+1).`
- template<Expression Ex>
  auto ceras::softsign (Ex const &ex) noexcept

  Softsign function, an unary operator. Returns `x / (abs(x) + 1).`
- template<Expression Ex>
  auto ceras::sigmoid (Ex const &ex) noexcept

  Sigmoid function, an unary operator. Returns `1 / (exp(-x) + 1).`
- template<Expression Ex>
  auto ceras::relu (Ex const &ex) noexcept

  Relu function, an unary operator. Returns `x` *if positive,* `0` *otherwise.*
- template<Expression Ex>
  auto ceras::relu6 (Ex const &ex) noexcept

  Rectified Linear 6 function, an unary operator. Returns `min(max(features, 0), 6).`
- template<typename T >
  requires std::floating_point<T>
  auto ceras::leaky_relu (T const factor=0.2) noexcept

  Leaky Rectified Linear function, an unary operator. Returns `x` *if positive,* `alpha x` *otherwise.* `alpha` *defaults to 0.2.*
- template<typename T >
  requires std::floating_point<T>
  auto ceras::prelu (T const factor) noexcept
- template<Expression Ex>
  auto ceras::negative_relu (Ex const &ex) noexcept
- template<typename T = float>
  requires std::floating_point<T>
  auto ceras::elu (T const alpha=1.0) noexcept

  Exponential Linear function, an unary operator. Returns `x` *if positive,* `alpha* (exp(x)-1)` *otherwise.* `alpha` *defaults to 0.2.*
- template<Expression Ex>
  auto ceras::exponential (Ex const &ex) noexcept

  Exponential function, an unary operator. Returns `exp(x).`
- template<Expression Ex>
  auto ceras::hard_sigmoid (Ex const &ex) noexcept

  Hard Sigmoid function, an unary operator. Piecewise linear approximation of the sigmoid function.
- template<Expression Ex>
  auto ceras::gelu (Ex const &ex) noexcept

  Gaussian Error function, an unary operator. GAUSSIAN ERROR LINEAR UNITS (GELUS) https://arxiv.org/pdf/1606.08415.pdf *$f(x) = 0.5x (1 + tanh[\sqrt{2/\pi}(x + 0.044715x^3)])$ $df = x (1 + tanh[\sqrt{2/\pi}(x + 0.044715x^3)]) + \sqrt(2/\pi) x sech^2[\sqrt(2/\pi) x (1+0.44715x^2) (1+0.134145x^2)]$ where $sec^2(x) = 1 - tanh^2(x)$ derivative generated using service from* https://www.symbolab.com/solver/derivative-calculator.
- template<Expression Ex>
  auto ceras::swish (Ex const &ex) noexcept

  Swish activation function.
- template<Expression Ex>
  auto ceras::silu (Ex const &ex) noexcept

  An alias name of activation swish.
- template<Expression Ex>
  auto ceras::crelu (Ex const &ex) noexcept

  Concatenated Rectified Linear Units, an activation function which preserves both positive and negative phase information while enforcing non-saturated non-linearity.

- template<Expression Ex>
  auto ceras::tank_shrink (Ex const &ex) noexcept

  Tank shrink function.
- template<Expression Ex>
  auto ceras::mish (Ex const &ex) noexcept

  Mish function.
- template<Expression Ex>
  auto ceras::lisht (Ex const &ex) noexcept

  Lisht function.

9.2 activation.hpp

Go to the documentation of this file.

```
#ifndef DJDWJBHNDAYTNOXLFOBDSGAQAAYPWMXJGEBYIRKEAKAQUUWVGDUGGDKSDXUKSPCYYNTWTDNII
#define DJDWJBHNDAYTNOXLFOBDSGAQAAYPWMXJGEBYIRKEAKAQUUWVGDUGGDKSDXUKSPCYYNTWTDNII

#include "./operation.hpp"
#include "./tensor.hpp"
#include "./utils/range.hpp"
#include "./utils/better_assert.hpp"
#include "./utils/for_each.hpp"
#include "./utils/context_cast.hpp"

namespace ceras
{

    template< std::floating_point Float >
    auto constexpr heaviside_step( Float f ) noexcept // f should not be zero
    {
        return [=]<Expression Ex>( Ex const& ex ) noexcept
        {
            return sigmoid( value( f+f ) * ex );
        };
    }

    // alias of heaviside_step(20)
    template <Expression Ex>
    auto constexpr soft_sign( Ex const& ex ) noexcept // soft-sign
    {
        return heaviside_step( 20.0 )( ex );
    }


    // alias of heaviside_step(20)
    template <Expression Ex>
    auto constexpr unit_step( Ex const& ex ) noexcept
    {
        return soft_sign( ex );
    }

    // alias of heaviside_step(20)
    template <Expression Ex>
    auto constexpr binary_step( Ex const& ex ) noexcept
    {
        return soft_sign( ex );
    }


    template <Expression Ex>
    auto constexpr gaussian( Ex const& ex ) noexcept
    {
        return exp( negative( square(ex) ) );
    }





    template <Expression Ex>
    auto constexpr softmax( Ex const& ex ) noexcept
    {
        return make_unary_operator( []<Tensor Tsor>( Tsor const& input ) noexcept
                                    {
                                        better_assert( !input.empty(), "softmax forward:  input tensor is
    empty!" );
                                        Tsor x = deep_copy( input );
```

```
                                    std::size_t const last_dim = *(x.shape().rbegin());
                                    std::size_t const rest_dim = x.size() / last_dim;
                                    for ( auto idx :  range( rest_dim ) )
                                    {
                                        auto [begin, end] = std::make_tuple( x.begin()+idx*last_dim, x.begin()+(idx+1)*last_dim );
                                        typename Tsor::value_type const mx = *std::max_element( begin, end );
                                        for_each( begin, end, [mx]( auto & v ){ v = std::exp( v-mx ); } );
                                        typename Tsor::value_type const sum = std::accumulate( begin, end, typename Tsor::value_type{0} );
                                        for_each( begin, end, [sum]( auto & v ){ v /= sum; } );
                                    }
                                    return x;
                                },
                                []<Tensor Tsor>( Tsor const&, Tsor const& output, Tsor const& grad ) noexcept
                                {
                                    better_assert( !has_nan( grad ), "backprop:  upcoming gradient for activation softmax contains NaN" );
                                    Tsor ans = grad;
                                    for_each( ans.begin(), ans.end(), output.begin(), []( auto& a, auto o ) { a *= o * ( typename Tsor::value_type{1} - o ); } );
                                    return ans;
                                },
                                "Softmax"
            )( ex );
  }

  template <Expression Ex>
  auto inline selu( Ex const& ex ) noexcept
  {
      std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
      std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();

      return make_unary_operator( [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                {
                                    typedef typename Tsor::value_type value_type;
                                    value_type const lambda = 1.0507;
                                    value_type const alpha = 1.67326;
                                    Tsor& ans = context_cast<Tsor>( forward_cache );
                                    ans.resize( input.shape() );
                                    std::copy( input.begin(), input.end(), ans.begin() );
                                    // if x >= 0:    \lambda x
                                    // if x <  0:    \lambda \alpha (exp(x) - 1)
                                    ans.map( [lambda, alpha](auto& x){ x = (x >= value_type{0}) ? (lambda * x) :  (lambda * alpha * (std::exp(x) - value_type{1})); } );
                                    return ans;
                                },
                                [backward_cache]<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor const& grad ) noexcept
                                {
                                    typedef typename Tsor::value_type value_type;
                                    value_type const lambda = 1.0507;
                                    value_type const alpha = 1.67326;
                                    Tsor& ans = context_cast<Tsor>( backward_cache );
                                    ans.resize( input.shape() ); // 1 / ( 1 + exp(-x) )
                                    // if x >= 0:  \lambda
                                    // if x <  0:  \lambda \alpha exp( x )
                                    for_each( ans.begin(), ans.end(), input.begin(), grad.begin(), [lambda, alpha]( auto& a, auto i, auto g ){ a = (i >= value_type{0}) ?  (g * lambda) :  (g * lambda * alpha * std::exp(i)); } );
                                    return ans;
                                },
                                "SeLU"
            )( ex );
  }

  template <Expression Ex>
  auto inline softplus( Ex const& ex ) noexcept
  {
      std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
      std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();

      return make_unary_operator( [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                {
                                    Tsor& ans = context_cast<Tsor>( forward_cache );
                                    ans.resize( input.shape() );
                                    std::copy( input.begin(), input.end(), ans.begin() );
                                    ans.map( [](auto& x){ x = std::log(1.0+std::exp(x)); } ); // ln( 1+e^x )
                                    return ans;
                                },
                                [backward_cache]<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor const& grad ) noexcept
                                {
```

```
                                        Tsor& ans = context_cast<Tsor>( backward_cache );
                                        ans.resize( input.shape() ); // 1 / ( 1 + exp(-x) )
                                        for_each( ans.begin(), ans.end(), input.begin(), grad.begin(),
[]( auto& a, auto i, auto g ){ a = g / ( typename Tsor::value_type{1} - std::exp(-i) ); } );
                                        return ans;
                                    },
                                    "SoftPlus"
                    )( ex );
    }


    template <Expression Ex>
    auto inline softsign( Ex const& ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();

        return make_unary_operator( [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( forward_cache );
                                        ans.resize( input.shape() );
                                        std::copy( input.begin(), input.end(), ans.begin() );
                                        ans.map( [](auto& x){ x /= typename Tsor::value_type{1} +
std::abs(x); } ); //  x / ( 1+|x| )
                                        return ans;
                                    },
                                    [backward_cache]<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor
const& grad ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( backward_cache );
                                        ans.resize( input.shape() ); // 1 / ( 1 + |x| )^2
                                        for_each( ans.begin(), ans.end(), input.begin(), grad.begin(),
[]( auto& a, auto i, auto g ){ auto tmp = typename Tsor::value_type{1} + std::abs(i); a = g /
(tmp*tmp); } );
                                        return ans;
                                    },
                                    "SoftSign"
                    )( ex );
    }

    template <Expression Ex>
    auto inline sigmoid( Ex const& ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();
        return make_unary_operator( [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( forward_cache );
                                        ans.resize( input.shape() );
                                        std::copy( input.begin(), input.end(), ans.begin() );
                                        //auto ans = input.deep_copy();
                                        ans.map( [](auto& x){ x = 1.0 / (1.0+std::exp(-x)); } );
                                        return ans;
                                    },
                                    [backward_cache]<Tensor Tsor>( Tsor const&, Tsor const& output, Tsor
const& grad ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( backward_cache );
                                        ans.resize( output.shape() );
                                        //Tsor ans{ output.shape() };
                                        for_each( ans.begin(), ans.end(), output.begin(), grad.begin(),
[]( auto & a, auto o, auto g ){ a = g * o * ( typename Tsor::value_type{1} - o ); } );
                                        return ans;
                                    },
                                    "Sigmoid"
                    )( ex );
    }


    namespace
    {
        struct relu_context
        {
            auto make_forward() const noexcept
            {
                return []( std::shared_ptr<std::any> forward_cache ) noexcept
                {
                    return [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                    {
                        typedef typename Tsor::value_type value_type;
                        Tsor& ans = context_cast<Tsor>( forward_cache );
                        ans.resize( input.shape()  );

                        for_each( ans.begin(), ans.end(), input.begin(), [](auto& o, auto x){ o =
std::max(x, value_type{0}); } );

                        return ans;
```

```
                };
            };
        }

        auto make_backward() const noexcept
        {
            return []<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor const& grad ) noexcept
            {
                typedef typename Tsor::value_type value_type;
                Tsor ans = grad; // shallow copy
                for_each( ans.begin(), ans.end(), input.begin(), []( auto& v, auto x ){ if ( x <= value_type{0} ) v = value_type{0}; } );
                return ans;
            };
        }
    }; // relu_context

}//anonymous namespace

template <Expression Ex>
auto relu( Ex const& ex ) noexcept
{
    std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
    return make_unary_operator( relu_context{}.make_forward()( forward_cache ), relu_context{}.make_backward(), "Relu")( ex );
}


namespace
{
    struct relu6_context
    {
        auto make_forward() const noexcept
        {
            return []( std::shared_ptr<std::any> forward_cache ) noexcept
            {
                return [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                {
                    typedef typename Tsor::value_type value_type;
                    Tsor& ans = context_cast<Tsor>( forward_cache );
                    ans.resize( input.shape()  );
                    for_each( ans.begin(), ans.end(), input.begin(), [](auto& o, auto x){ o = std::min( value_type{6}, std::max(x, value_type{0}) ); } );
                    return ans;
                };
            };
        }

        auto make_backward() const noexcept
        {
            return []<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor const& grad ) noexcept
            {
                typedef typename Tsor::value_type value_type;
                Tsor ans = grad; // shallow copy
                //const typename Tsor::value_type zero{0};
                for_each( ans.begin(), ans.end(), input.begin(), []( auto& v, auto x ){ if ( (x <= value_type{0}) || (x >= value_type{6}) ) v = value_type{0}; } );
                return ans;
            };
        }
    }; // relu6_context

}//anonymous namespace

template <Expression Ex>
auto relu6( Ex const& ex ) noexcept
{
    std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
    return make_unary_operator( relu6_context{}.make_forward()( forward_cache ), relu6_context{}.make_backward(), "Relu6")( ex );
}


template< typename T > requires std::floating_point<T>
auto leaky_relu( T const factor=0.2 ) noexcept
{
    better_assert( factor > T{0}, "Expecting leak_relu with a factor greater than 0, but got factor = ", factor );
    better_assert( factor < T{1}, "Expecting leak_relu with a factor less than 1, but got factor = ", factor );
    return [factor]<Expression Ex>( Ex const& ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        return make_unary_operator( [factor, forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( forward_cache );
```

```
                                        ans.resize( input.shape()  );
                                        for_each( ans.begin(), ans.end(), input.begin(), [factor]( auto& v_out, auto v_in ){ v_out = std::max( T{v_in}, T{factor*v_in} ); } );
                                        return ans;
                                    },
                                    [factor]<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor const& grad ) noexcept
                                    {
                                        typedef typename Tsor::value_type value_type;
                                        Tsor ans = grad;// OK for shallow copy
                                        for_each( ans.begin(), ans.end(), input.begin(), [factor]( value_type& v_back, value_type const v_in ){ v_back = (v_in > value_type{0}) ?  v_back : factor*v_back; } );
                                        return ans;
                                    },
                                    "LeakyRelu"
                    )( ex );
        };
    }

    template< typename T > requires std::floating_point<T>
    auto prelu( T const factor ) noexcept
    {
        return leaky_relu( factor );
    }

    template <Expression Ex>
    auto negative_relu( Ex const& ex ) noexcept
    {
        return negative( relu( ex ) );
    }


    template< typename T=float > requires std::floating_point<T>
    auto elu( T const alpha=1.0 ) noexcept
    {
        return [alpha]<Expression Ex>( Ex const& ex ) noexcept
        {
            std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
            return make_unary_operator( [alpha, forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                    {
                                        typedef typename Tsor::value_type value_type;
                                        Tsor& ans = context_cast<Tsor>( forward_cache );
                                        ans.resize( input.shape()  );
                                        for_each( ans.begin(), ans.end(), input.begin(), [alpha]( auto& v_out, auto v_in ){ v_out = (v_in > value_type{0}) ?  v_in :  (alpha * (std::exp(v_in) - value_type{1})); } );
                                        return ans;
                                    },
                                    [alpha]<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor const& grad ) noexcept
                                    {
                                        typedef typename Tsor::value_type value_type;
                                        Tsor ans = grad;// OK for shallow copy
                                        for_each( ans.begin(), ans.end(), input.begin(), [alpha]( value_type& v_back, value_type const v_in ){ v_back = (v_in >= value_type{0}) ?  v_back : alpha*std::exp(v_back); } );
                                        return ans;
                                    },
                                    "ELU"
                    )( ex );
        };
    }

    template <Expression Ex>
    auto inline exponential( Ex const& ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();

        return make_unary_operator( [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                {
                                    Tsor& ans = context_cast<Tsor>( forward_cache );
                                    ans.resize( input.shape() );
                                    std::copy( input.begin(), input.end(), ans.begin() );
                                    ans.map( [](auto& x){ x = std::exp(x); } ); // exp(x)
                                    better_assert( !has_nan( ans ), "exponential operator forward output contains nan." );
                                    better_assert( !has_inf( ans ), "exponential operator forward output contains inf." );
                                    return ans;
                                },
                                []<Tensor Tsor>( Tsor const&, Tsor const& output, Tsor const& grad ) noexcept
                                {
                                    Tsor ans = grad;
                                    for_each( ans.begin(), ans.end(), output.begin(), []( auto& a,
```

```
auto o ){ a *= o; } );
                                      return ans;
                                  },
                                  "Exponentional"
               )( ex );
  }

  template <Expression Ex>
  auto inline hard_sigmoid( Ex const& ex ) noexcept
  {
      std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();

      return make_unary_operator( [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                  {
                                      typedef typename Tsor::value_type value_type;
                                      Tsor& ans = context_cast<Tsor>( forward_cache );
                                      ans.resize( input.shape() );
                                      std::copy( input.begin(), input.end(), ans.begin() );
                                      ans.map([](auto& x) { x = ( x > value_type{1} ) ? value_type{1} : ( x < value_type{-1} ) ? value_type{0} : (x+value_type{1})/value_type{2}; });
                                      return ans;
                                  },
                                  []<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor const& grad ) noexcept
                                  {
                                      typedef typename Tsor::value_type value_type;
                                      Tsor ans = grad;
                                      for_each( ans.begin(), ans.end(), input.begin(), []( auto& a, auto x ) { a = ((x > value_type{1}) || (x < value_type{-1})) ? value_type{0} : (a / value_type{2}); } );
                                      return ans;
                                  },
                                  "HardSigmoid"
               )( ex );
  }

  template <Expression Ex>
  auto inline gelu( Ex const& ex ) noexcept
  {
      auto _gelu = []<typename T>( T x )
      {
          auto const ans = 0.5 * x * ( 1.0 + std::tanh( 0.79788456080286535588 * x ( 1.0 + 0.044715*x*x ) ) );
          return static_cast<T>( ans );
      };
      auto sech_2 = []( auto x )
      {
          return 1.0 - std::pow( std::tanh(x), 2 );
      };
      auto _dgelu = [sech_2]<typename T>( T x )
      {
          auto const sq_2_pi_x = 0.79788456080286535588 * x;
          auto const _xx = x * x;
          auto const ans = 0.5 * ( 1.0 + std::tanh( sq_2_pi_x * ( 1.0 + 0.044715 * _xx ) ) ) + sq_2_pi_x * sech_2( sq_2_pi_x * (1.0 + 0.044715 * _xx ) * ( 1.0 + 0.134145 * _xx) );
          return static_cast<T>( ans );
      };

      std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();

      return make_unary_operator( [forward_cache, _gelu]<Tensor Tsor>( Tsor const& input ) noexcept
                                  {
                                      //typedef typename Tsor::value_type value_type;
                                      Tsor& ans = context_cast<Tsor>( forward_cache );
                                      ans.resize( input.shape() );
                                      std::copy( input.begin(), input.end(), ans.begin() );
                                      ans.map([_gelu](auto& x) { x = _gelu(x); });
                                      return ans;
                                  },
                                  [_dgelu]<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor const& grad ) noexcept
                                  {
                                      //typedef typename Tsor::value_type value_type;
                                      Tsor ans = grad;
                                      for_each( ans.begin(), ans.end(), [&_dgelu]( auto& x ) { x = _dgelu(x); } );
                                      return ans;
                                  },
                                  "GeLU"
               )( ex );
  }


  template< Expression Ex >
  auto swish( Ex const& ex ) noexcept
  {
      return hadamard_product( ex, sigmoid( ex ) );
```

```
    }

    template< Expression Ex >
    auto silu( Ex const& ex ) noexcept
    {
        return swish( ex );
    }

    template< Expression Ex >
    auto crelu( Ex const& ex ) noexcept
    {
        return concatenate(-1)( relu(ex), relu(-ex) );
    }

    template< Expression Ex >
    auto tank_shrink( Ex const& ex ) noexcept
    {
        return ex - tanh( ex );
    }


    template< Expression Ex >
    auto mish( Ex const& ex ) noexcept
    {
        return ex*tanh(softplus(ex));
    }


    template< Expression Ex >
    auto lisht( Ex const& ex ) noexcept
    {
        return ex*tanh(ex);
    }

}//namespace ceras

#endif//DJDWJBHNDAYTNOXLFOBDSGAQAAYPWMXJGEBYIRKEAKAQUUWVGDUGGDKSDXUKSPCYYNTWTDNII

```

9.3 /home/feng/workspace/github.repo/ceras/include/ceras.hpp File Reference

```
#include "./config.hpp"
#include "./includes.hpp"
#include "./activation.hpp"
#include "./ceras.hpp"
#include "./loss.hpp"
#include "./operation.hpp"
#include "./complex_operator.hpp"
#include "./optimizer.hpp"
#include "./place_holder.hpp"
#include "./session.hpp"
#include "./tensor.hpp"
#include "./variable.hpp"
#include "./constant.hpp"
#include "./layer.hpp"
#include "./model.hpp"
#include "./dataset.hpp"
```

9.4 ceras.hpp

Go to the documentation of this file.

```
#ifndef AOXWQXCBBVJUJJSDTMLRNOHUNBOARCNCRGBTAHMIKSULFJOIUGCEEGXXVRUVWTOSKPYWDJOKQ
#define AOXWQXCBBVJUJJSDTMLRNOHUNBOARCNCRGBTAHMIKSULFJOIUGCEEGXXVRUVWTOSKPYWDJOKQ

```

```
#include "./config.hpp"
#include "./includes.hpp"
#include "./activation.hpp"
#include "./ceras.hpp"
#include "./config.hpp"
#include "./loss.hpp"
#include "./operation.hpp"
#include "./complex_operator.hpp"
#include "./optimizer.hpp"
#include "./place_holder.hpp"
#include "./session.hpp"
#include "./tensor.hpp"
#include "./variable.hpp"
#include "./constant.hpp"
#include "./layer.hpp"
#include "./model.hpp"
#include "./dataset.hpp"

#endif//AOXWQXCBBVJUJJSDTMLRNOHUNBOARCNCRGBTAHMIKSULFJOIUGCEEGXXVRUVWTOSKPYWDJOKQ

```

9.5 /home/feng/workspace/github.repo/ceras/include/complex_↩ operator.hpp File Reference

```
#include "./operation.hpp"
```

Classes

- struct ceras::complex< Real_Ex, Imag_Ex >
- struct ceras::is_complex< T >
- struct ceras::is_complex< complex< Real_Ex, Imag_Ex > >

Namespaces

- namespace ceras

Concepts

- concept ceras::Complex

  A type that represents a complex expression.

Functions

- template< Expression Real_Ex, Expression Imag_Ex>
  Real_Ex ceras::real (complex< Real_Ex, Imag_Ex > const &c) noexcept
- template< Expression Real_Ex, Expression Imag_Ex>
  Imag_Ex ceras::imag (complex< Real_Ex, Imag_Ex > const &c) noexcept
- template< Complex C>
  auto ceras::abs (C const &c) noexcept

  Returns the magnitude of the complex expression.
- template< Complex C>
  auto ceras::norm (C const &c) noexcept

  Returns the squared magnitude of the complex expression.

- template<Complex C>
  auto ceras::conj (C const &c) noexcept

  Returns the conjugate of the complex expression.
- template<Expression Em, Expression Ep>
  auto ceras::polar (Em const &em, Ep const &ep) noexcept

  Returns with given magnitude and phase angle.
- template<Complex C>
  auto ceras::arg (C const &c) noexcept

  Calculates the phase angle (in radians) of the complex expression.
- template<Complex C>
  auto ceras::operator+ (C const &c) noexcept

  Returns the complex expression.
- template<Complex C>
  auto ceras::operator- (C const &c) noexcept

  Negatives the complex expression.
- template<Complex Cl, Complex Cr>
  auto ceras::operator+ (Cl const &cl, Cr const &cr) noexcept

  Sums up two complex expressions.
- template<Complex Cl, Complex Cr>
  auto ceras::operator- (Cl const &cl, Cr const &cr) noexcept

  Subtracts one complex expression from the other one.
- template<Complex Cl, Complex Cr>
  auto ceras::operator* (Cl const &cl, Cr const &cr) noexcept

  Multiplies two complex expressions. Optimization here: (a+ib)(c+id) = (ac-bd) + i(ad+bc) = (ac-bd) + i((a+b)*(c+d)-ac-bd)*
- template<Complex C, Expression E>
  auto ceras::operator+ (C const &c, E const &e) noexcept

  Sums up a complex expression and an expression.
- template<Complex C, Expression E>
  auto ceras::operator+ (E const &e, C const &c) noexcept

  Sums up a complex expression and an expression.
- template<Complex C, Expression E>
  auto ceras::operator- (C const &c, E const &e) noexcept

  Subtracts an expression from a compression expression.
- template<Complex C, Expression E>
  auto ceras::operator- (E const &e, C const &c) noexcept

  Subtractsa complex expression from an expression.
- template<Complex C, Expression E>
  auto ceras::operator* (C const &c, E const &e) noexcept

  Multiplies a complex expression with an expression.
- template<Complex C, Expression E>
  auto ceras::operator* (E const &e, C const &c) noexcept

  Multiplies an expression with a compression expression.

Variables

- template<typename T >
  constexpr bool ceras::is_complex_v = is_complex<T>::value

9.6 complex_operator.hpp

Go to the documentation of this file.

```
#ifndef SJTXPMDBEYHNPQSGKFIEKTKOYWOFMOEGAHNOHMJVHJIAWTBHCCFUKCHLJMJAFPRHRXEOTYEDC
#define SJTXPMDBEYHNPQSGKFIEKTKOYWOFMOEGAHNOHMJVHJIAWTBHCCFUKCHLJMJAFPRHRXEOTYEDC

#include "./operation.hpp"

namespace ceras
{

    template< Expression Real_Ex, Expression Imag_Ex >
    struct complex
    {
        Real_Ex real_;
        Imag_Ex imag_;
    };//struct complex


    template< typename T >
    struct is_complex :  std::false_type {};

    template< Expression Real_Ex, Expression Imag_Ex >
    struct is_complex<complex<Real_Ex, Imag_Ex» :  std::true_type {};

    template< typename T >
    constexpr bool is_complex_v = is_complex<T>::value;

    template< typename T >
    concept Complex = is_complex_v<T>;


    template< Expression Real_Ex, Expression Imag_Ex >
    Real_Ex real( complex<Real_Ex, Imag_Ex> const& c ) noexcept
    {
        return c.real_;
    }

    template< Expression Real_Ex, Expression Imag_Ex >
    Imag_Ex imag( complex<Real_Ex, Imag_Ex> const& c ) noexcept
    {
        return c.imag_;
    }


    template< Complex C >
    auto abs( C const& c ) noexcept
    {
        return hypot( real(c), imag(c) );
    }


    template< Complex C >
    auto norm( C const& c ) noexcept
    {
        auto const& r = real( c );
        auto const& i = imag( c );
        return hadamard_product( r, r ) + hadamard_product( i, i );
    }


    template< Complex C >
    auto conj( C const& c ) noexcept
    {
        return complex{ real(c), -imag(c) };
    }


    template< Expression Em, Expression Ep >
    auto polar( Em const& em, Ep const& ep ) noexcept
    {
        return complex{ hadamard_product( em, cos(ep) ), hadamard_product( em, sin(ep) ) };
    }



    template< Complex C >
    auto arg( C const& c ) noexcept
    {
        return atan2( imag(c), real(c) );
    }


    template< Complex C >
    auto operator + ( C const& c ) noexcept
```

```
    {
        return c;
    }

    template< Complex C >
    auto operator - ( C const& c ) noexcept
    {
        return complex{ negative(real(c)), negative(imag(c)) };
    }


    template< Complex Cl, Complex Cr >
    auto operator + ( Cl const& cl, Cr const& cr ) noexcept
    {
        return complex{ real(cl)+real(cr), imag(cl)+imag(cr) };
    }


    template< Complex Cl, Complex Cr >
    auto operator - ( Cl const& cl, Cr const& cr ) noexcept
    {
        return complex{ real(cl)-real(cr), imag(cl)-imag(cr) };
    }


    template< Complex Cl, Complex Cr >
    auto operator * ( Cl const& cl, Cr const& cr ) noexcept
    {
        auto const& a = real(cl);
        auto const& b = imag(cl);
        auto const& c = real(cr);
        auto const& d = imag(cr);
        auto const& ac = a * c;          // 1st multiplication
        auto const& bd = b * d;          // 2nd multiplication
        auto const& a_b = a + b;
        auto const& c_d = c + d;
        auto const& abcd = a_b * c_d;    // 3rd multiplication

        return complex{ ac-bd, abcd-ac-bd };
    }



    template< Complex C, Expression E >
    auto operator + ( C const& c, E const& e ) noexcept
    {
        return complex{ real(c)+e, imag(c) };
    }

    template< Complex C, Expression E >
    auto operator + ( E const& e, C const& c ) noexcept
    {
        return c + e;
    }


    template< Complex C, Expression E >
    auto operator - ( C const& c, E const& e ) noexcept
    {
        return complex{ real(c)-e, imag(c) };
    }

    template< Complex C, Expression E >
    auto operator - ( E const& e, C const& c ) noexcept
    {
        return c + e;
    }

    template< Complex C, Expression E >
    auto operator * ( C const& c, E const& e ) noexcept
    {
        return complex{ real(c)*e, imag(c)*e };
    }

    template< Complex C, Expression E >
    auto operator * ( E const& e, C const& c ) noexcept
    {
        return c * e;
    }


}//namespace ceras

#endif//SJTXPMDBEYHNPQSGKFIEKTKOYWOFMOEGAHNOHMJVHJIAWTBHCCFUKCHLJMJAFPRHRXEOTYEDC

```

9.7 /home/feng/workspace/github.repo/ceras/include/config.hpp File Reference

9.8 config.hpp

Go to the documentation of this file.

```
#ifndef FBXYAXRPGSNHIXESHOGNYHPVEWWVSRSRJLQPIRIFENBGNMGFLJNMWXDNQLHKOAGBNYGBJRLBD
#define FBXYAXRPGSNHIXESHOGNYHPVEWWVSRSRJLQPIRIFENBGNMGFLJNMWXDNQLHKOAGBNYGBJRLBD

namespace ceras
{
    inline constexpr unsigned long version = 20211219UL;
    inline constexpr unsigned long __version__ = version;

#ifdef _MSC_VER
        inline constexpr unsigned long is_windows_platform = 1;
#else
        inline constexpr unsigned long is_windows_platform = 0;
#endif

#ifdef NDEBUG
        inline constexpr unsigned long debug_mode = 0;
#else
        inline constexpr unsigned long debug_mode = 1;
#endif

#ifdef CBLAS
        inline constexpr unsigned long cblas_mode = 1;
#else
        inline constexpr unsigned long cblas_mode = 0;
#endif

#ifndef NOPARALLEL
        inline constexpr unsigned long parallel_mode = 1;
#else
        inline constexpr unsigned long parallel_mode = 0;
#endif

#ifdef CUDA
        inline constexpr unsigned long cuda_mode = 1;
#else
        inline constexpr unsigned long cuda_mode = 0;
#endif

    inline int visible_device = 0; // using GPU 0 by default
    inline unsigned long cuda_gemm_threshold = 0UL; // will be updated if in CUDA mode, always assume float multiplications as double is rearly used

    inline constexpr double eps = 1.0e-8;
    inline constexpr double epsilon = eps; // alias of ‘eps‘

    inline int learning_phase = 1;
}

#endif//FBXYAXRPGSNHIXESHOGNYHPVEWWVSRSRJLQPIRIFENBGNMGFLJNMWXDNQLHKOAGBNYGBJRLBD

```

9.9 /home/feng/workspace/github.repo/ceras/include/constant.hpp File Reference

```
#include "./includes.hpp"
#include "./tensor.hpp"
#include "./utils/id.hpp"
#include "./utils/better_assert.hpp"
#include "./utils/enable_shared.hpp"
```

Classes

- struct ceras::constant< Tsor >

    Creates a constant expression from a tensor-like object.
- struct ceras::is_constant< T >
- struct ceras::is_constant< constant< Tsor > >

Namespaces

- namespace ceras

Concepts

- concept ceras::Constant

Variables

- template<class T >
  constexpr bool ceras::is_constant_v = is_constant<T>::value

9.10 constant.hpp

Go to the documentation of this file.

```
#ifndef CONSTANT_HPP_INCLUDED_DLKJASLKJFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
#define CONSTANT_HPP_INCLUDED_DLKJASLKJFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

#include "./includes.hpp"
#include "./tensor.hpp"
#include "./utils/id.hpp"
#include "./utils/better_assert.hpp"
#include "./utils/enable_shared.hpp"

namespace ceras
{

    template< Tensor Tsor >
    struct constant :  enable_id<constant<Tsor>, "Constant">
    {
        typedef Tsor tensor_type;
        tensor_type data_;

        constant( tensor_type const& data ) :  enable_id<constant<tensor_type>, "Constant">{},
    data_{data} {}

        void backward( auto )const {}

        tensor_type forward()const
{
            return data_;
        }

        auto shape()const
{
            return data_.shape();
        }
    };

    template< typename T >
    struct is_constant :  std::false_type {};

    template< Tensor Tsor >
    struct is_constant< constant< Tsor > > :  std::true_type {};

    template< class T >
    inline constexpr bool is_constant_v = is_constant<T>::value;

    template< typename T >
    concept Constant = is_constant_v<T>;

}//namespace ceras

#endif//CONSTANT_HPP_INCLUDED_DLKJASLKJFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

```

9.11 /home/feng/workspace/github.repo/ceras/include/dataset.hpp File Reference

```
#include "./tensor.hpp"
#include "./includes.hpp"
#include "./utils/better_assert.hpp"
#include "./utils/for_each.hpp"
```

Namespaces

- namespace ceras
- namespace ceras::dataset
- namespace ceras::dataset::mnist
- namespace ceras::dataset::fashion_mnist

Functions

- auto ceras::dataset::mnist::load_data (std::string const &path=std::string{"./dataset/mnist"})
- auto ceras::dataset::fashion_mnist::load_data (std::string const &path=std::string{"./dataset/fashion_mnist"})

9.12 dataset.hpp

Go to the documentation of this file.

```
#ifndef RKQSLRMXHSPFGGPQCNEPEBAKCXHNXQPMXETNTTXBWEWBIQHVCFRKRFSFMLXXXRYFUKHEXYIGL
#define RKQSLRMXHSPFGGPQCNEPEBAKCXHNXQPMXETNTTXBWEWBIQHVCFRKRFSFMLXXXRYFUKHEXYIGL

#include "./tensor.hpp"
#include "./includes.hpp"
#include "./utils/better_assert.hpp"
#include "./utils/for_each.hpp"

namespace ceras::dataset
{

    namespace mnist
    {
        inline auto load_data( std::string const& path = std::string{"./dataset/mnist"} )
        {
            std::string const training_image_path = path + std::string{"/train-images-idx3-ubyte"};
            std::string const training_label_path = path + std::string{"/train-labels-idx1-ubyte"};
            std::string const test_image_path = path + std::string{"/t10k-images-idx3-ubyte"};
            std::string const test_label_path = path + std::string{"/t10k-labels-idx1-ubyte"};

            auto const& load_binary = []( std::string const& filename )
            {
                std::ifstream ifs( filename, std::ios::binary );
                better_assert( ifs.good(), "Failed to load data from ", filename );
                std::vector<char> buff{ ( std::istreambuf_iterator<char>( ifs ) ), ( std::istreambuf_iterator<char>() ) };
                std::vector<std::uint8_t> ans( buff.size() );
                std::copy( buff.begin(), buff.end(), reinterpret_cast<char*>( ans.data() ) );
                return ans;
            };

            auto const& extract_image = []( std::vector<std::uint8_t> const& image_data )
            {
                unsigned long const offset = 16;
                unsigned long const samples = (image_data.size()-offset) / (28*28);
                tensor<std::uint8_t> ans{ {samples, 28, 28} };
                std::copy( image_data.begin()+offset, image_data.end(), ans.data() );
                return ans;
            };

            auto const& extract_label = []( std::vector<std::uint8_t> const& label_data )
```

```
            {
                unsigned long const offset = 8;
                unsigned long const samples = label_data.size() - offset;
                auto ans = zeros<std::uint8_t>({samples, 10});
                auto ans_2d = matrix_view{ ans.data(), samples, 10 };
                for ( auto idx :  range( samples ) )
                    ans_2d[idx][label_data[idx+offset]] = 1;
                return ans;
            };

            return std::make_tuple( extract_image(load_binary(training_image_path)),
                                    extract_label(load_binary(training_label_path)),
                                    extract_image(load_binary(test_image_path)),
                                    extract_label(load_binary(test_label_path)) );
        }
    }

    namespace fashion_mnist
    {
        inline auto load_data( std::string const& path = std::string{"./dataset/fashion_mnist"} )
        {
            std::string const training_image_path = path + std::string{"/train-images-idx3-ubyte"};
            std::string const training_label_path = path + std::string{"/train-labels-idx1-ubyte"};
            std::string const test_image_path = path + std::string{"/t10k-images-idx3-ubyte"};
            std::string const test_label_path = path + std::string{"/t10k-labels-idx1-ubyte"};

            auto const& load_binary = []( std::string const& filename )
            {
                std::ifstream ifs( filename, std::ios::binary );
                better_assert( ifs.good(), "Failed to load data from ", filename );
                std::vector<char> buff{ ( std::istreambuf_iterator<char>( ifs ) ), ( std::istreambuf_iterator<char>() ) };
                std::vector<std::uint8_t> ans( buff.size() );
                std::copy( buff.begin(), buff.end(), reinterpret_cast<char*>( ans.data() ) );
                return ans;
            };

            auto const& extract_image = []( std::vector<std::uint8_t> const& image_data )
            {
                unsigned long const offset = 16;
                unsigned long const samples = (image_data.size()-offset) / (28*28);
                tensor<std::uint8_t> ans{ {samples, 28, 28} };
                std::copy( image_data.begin()+offset, image_data.end(), ans.data() );
                return ans;
            };

            auto const& extract_label = []( std::vector<std::uint8_t> const& label_data )
            {
                unsigned long const offset = 8;
                unsigned long const samples = label_data.size() - offset;
                auto ans = zeros<std::uint8_t>({samples, 10});
                auto ans_2d = matrix_view{ ans.data(), samples, 10 };
                for ( auto idx :  range( samples ) )
                    ans_2d[idx][label_data[idx+offset]] = 1;
                return ans;
            };

            return std::make_tuple( extract_image(load_binary(training_image_path)),
                                    extract_label(load_binary(training_label_path)),
                                    extract_image(load_binary(test_image_path)),
                                    extract_label(load_binary(test_label_path)) );
        }//load_data

    }//fashion_mnist


#if 0
    namespace cifar10
    {
        inline auto load_data( std::string const& path = std::string{} )
        {
        }
    }

    namespace cifar100
    {
        inline auto load_data( std::string const& path = std::string{} )
        {
        }
    }

    namespace imdb
    {
        inline auto load_data( std::string const& path = std::string{} )
        {
        }
    }
```

```
    namespace reuters
    {
        inline auto load_data( std::string const& path = std::string{} )
        {
        }
    }

    namespace boston_housing
    {
        inline auto load_data( std::string const& path = std::string{} )
        {
        }
    }
#endif


}//namespace ceras

#endif//RKQSLRMXHSPFGGPQCNEPEBAKCXHNXQPMXETNTTXBWEWBIQHVCFRKRFSFMLXXXRYFUKHEXYIGL

```

9.13 /home/feng/workspace/github.repo/ceras/include/includes.hpp File Reference

```
#include "./config.hpp"
#include <algorithm>
#include <any>
#include <array>
#include <cassert>
#include <chrono>
#include <climits>
#include <cmath>
#include <compare>
#include <concepts>
#include <cstddef>
#include <cstdint>
#include <cstdlib>
#include <cstdio>
#include <ctime>
#include <exception>
#include <filesystem>
#include <fstream>
#include <functional>
#include <initializer_list>
#include <iomanip>
#include <ios>
#include <iostream>
#include <istream>
#include <iterator>
#include <limits>
#include <map>
#include <memory>
#include <numeric>
#include <optional>
#include <ostream>
#include <random>
#include <ranges>
#include <regex>
#include <set>
#include <sstream>
#include <string>
```

```
#include <thread>
#include <tuple>
#include <type_traits>
#include <unordered_map>
#include <unordered_set>
#include <utility>
#include <vector>
#include "./utils/3rd_party/stb_image.h"
#include "./utils/3rd_party/stb_image_write.h"
#include "./utils/3rd_party/stb_image_resize.h"
#include "./utils/3rd_party/glob.hpp"
```

Macros

- #define STB_IMAGE_IMPLEMENTATION
- #define STB_IMAGE_WRITE_IMPLEMENTATION
- #define STB_IMAGE_RESIZE_IMPLEMENTATION

9.13.1 Macro Definition Documentation

9.13.1.1 STB_IMAGE_IMPLEMENTATION

```
#define STB_IMAGE_IMPLEMENTATION
```

9.13.1.2 STB_IMAGE_RESIZE_IMPLEMENTATION

```
#define STB_IMAGE_RESIZE_IMPLEMENTATION
```

9.13.1.3 STB_IMAGE_WRITE_IMPLEMENTATION

```
#define STB_IMAGE_WRITE_IMPLEMENTATION
```

9.14 includes.hpp

Go to the documentation of this file.

```
#ifndef JIDOKQFMGRFETGICWFQXOAUHPAYLQWXAGIBPFTAXKFLOWQMWEAWUOKJJBUNTLLMIHYSFJQOYA
#define JIDOKQFMGRFETGICWFQXOAUHPAYLQWXAGIBPFTAXKFLOWQMWEAWUOKJJBUNTLLMIHYSFJQOYA

#include "./config.hpp"

#include <algorithm>
#include <any>
#include <array>
#include <cassert>
#include <chrono>
#include <climits>
#include <cmath>
#include <compare>
#include <concepts>
#include <cstddef>
#include <cstdint>
#include <cstdlib>
#include <cstdio>
#include <ctime>
#include <exception>
#include <filesystem>
#include <fstream>
#include <functional>
#include <initializer_list>
#include <iomanip>
#include <ios>
#include <iostream>
#include <istream>
#include <iterator>
#include <limits>
#include <map>
#include <memory>
#include <numeric>
#include <optional>
#include <ostream>
#include <random>
#include <ranges>
#include <regex>
#include <set>
#include <sstream>
#include <string>
#include <thread>
#include <tuple>
#include <type_traits>
#include <unordered_map>
#include <unordered_set>
#include <utility>
#include <vector>

//
// begin of 3rd party libraries
//

#define STB_IMAGE_IMPLEMENTATION
#include "./utils/3rd_party/stb_image.h"
#define STB_IMAGE_WRITE_IMPLEMENTATION
#include "./utils/3rd_party/stb_image_write.h"
#define STB_IMAGE_RESIZE_IMPLEMENTATION
#include "./utils/3rd_party/stb_image_resize.h"

#include "./utils/3rd_party/glob.hpp"

//
// end of 3rd party libraries
//

#endif//JIDOKQFMGRFETGICWFQXOAUHPAYLQWXAGIBPFTAXKFLOWQMWEAWUOKJJBUNTLLMIHYSFJQOYA
```

9.15 /home/feng/workspace/github.repo/ceras/include/layer.hpp File Reference

```
#include "./operation.hpp"
#include "./activation.hpp"
```

```
#include "./loss.hpp"
#include "./optimizer.hpp"
#include "./utils/better_assert.hpp"
```

Namespaces

- namespace ceras

Functions

- auto ceras::Input (std::vector< unsigned long > const &input_shape={{-1UL}})
- auto ceras::Conv2D (unsigned long output_channels, std::vector< unsigned long > const &kernel_size, std::vector< unsigned long > const &input_shape, std::string const &padding="valid", std::vector< unsigned long > const &strides={1, 1}, std::vector< unsigned long > const &dilations={1, 1}, bool use_bias=true, float kernel_regularizer_l1=0.0f, float kernel_regularizer_l2=0.0f, float bias_regularizer_l1=0.0f, float bias_regularizer_l2=0.0f) noexcept

    2D convolution layer.
- auto ceras::Conv2D (unsigned long output_channels, std::vector< unsigned long > const &kernel_size, std::string const &padding="valid", std::vector< unsigned long > const &strides={1, 1}, std::vector< unsigned long > const &dilations={1, 1}, bool use_bias=true, float kernel_regularizer_l1=0.0f, float kernel_regularizer_l2=0.0f, float bias_regularizer_l1=0.0f, float bias_regularizer_l2=0.0f) noexcept

    2D convolution layer.
- auto ceras::Dense (unsigned long output_size, bool use_bias=true, float kernel_regularizer_l1=0.0f, float kernel_regularizer_l2=0.0f, float bias_regularizer_l1=0.0f, float bias_regularizer_l2=0.0f)

    Densly-connected layer.
- auto ceras::BatchNormalization (float threshold=0.95f, float kernel_regularizer_l1=0.0f, float kernel_regularizer_l2=0.0f, float bias_regularizer_l1=0.0f, float bias_regularizer_l2=0.0f)

    Applies a transformation that maintains the mean output close to 0 and the output standard deviation close to 1.
- auto ceras::Concatenate (unsigned long axis=-1) noexcept
- auto ceras::Add () noexcept
- auto ceras::Subtract () noexcept
- auto ceras::Multiply () noexcept
- template<Expression Ex>
  auto ceras::ReLU (Ex const &ex) noexcept
- auto ceras::Softmax () noexcept
- template<typename T = float>
  auto ceras::LeakyReLU (T const factor=0.2) noexcept
- template<typename T = float>
  auto ceras::ELU (T const factor=0.2) noexcept
- auto ceras::Reshape (std::vector< unsigned long > const &new_shape, bool include_batch_flag=true) noexcept
- auto ceras::Flatten () noexcept
- auto ceras::MaxPooling2D (unsigned long stride) noexcept
- auto ceras::UpSampling2D (unsigned long stride) noexcept
- template<typename T >
  auto ceras::Dropout (T factor) noexcept
- auto ceras::AveragePooling2D (unsigned long stride) noexcept

9.16 layer.hpp

Go to the documentation of this file.

```
#ifndef NLESIGQPSASUTOXPLGXCUHFGGUGYSWLQQFATNISJOSPUFHRORXBNXLSWTYRNSIWJKYFXIQXVN
#define NLESIGQPSASUTOXPLGXCUHFGGUGYSWLQQFATNISJOSPUFHRORXBNXLSWTYRNSIWJKYFXIQXVN

#include "./operation.hpp"
#include "./activation.hpp"
#include "./loss.hpp"
#include "./optimizer.hpp"
#include "./utils/better_assert.hpp"

// try to mimic classes defined in tensorflow.keras

namespace ceras
{


    inline auto Input( std::vector<unsigned long> const& input_shape = {{-1UL}} )
    {
        return place_holder<tensor<float»{ input_shape };
    }

    [[deprecated("input_shape is not required in the new Conv2D(), this interface will be removed.")]]
    inline auto Conv2D( unsigned long output_channels, std::vector<unsigned long> const& kernel_size,
                        std::vector<unsigned long> const& input_shape, std::string const& padding="valid",
                        std::vector<unsigned long> const& strides={1,1}, std::vector<unsigned long> const& dilations={1, 1}, bool use_bias=true,
                        float kernel_regularizer_l1=0.0f, float kernel_regularizer_l2=0.0f, float bias_regularizer_l1=0.0f, float bias_regularizer_l2=0.0f
            ) noexcept
    {
        better_assert( output_channels > 0, "Expecting output_channels larger than 0." );
        better_assert( kernel_size.size() > 0, "Expecting kernel_size at least has 1 elements." );
        better_assert( input_shape.size() ==3, "Expecting input_shape has 3 elements." );
        better_assert( strides.size() > 0, "Expecting strides at least has 1 elements." );
        return [=]<Expression Ex>( Ex const& ex ) noexcept
        {
            unsigned long const kernel_size_x = kernel_size[0];
            unsigned long const kernel_size_y = kernel_size.size() == 2 ?  kernel_size[1] : kernel_size[0];
            //unsigned long const kernel_size_y = kernel_size[1];
            unsigned long const input_channels = input_shape[2];
            unsigned long const input_x = input_shape[0];
            unsigned long const input_y = input_shape[1];
            unsigned long const stride_x = strides[0];
            unsigned long const stride_y = strides.size() == 2 ?  strides[1] :  strides[0];
            unsigned long const dilation_row = dilations[0];
            unsigned long const dilation_col = dilations.size() == 2 ?  dilations[1] :  dilations[0];
            //unsigned long const stride_y = strides[1];
            auto w = variable<tensor<float»{ glorot_uniform<float>({output_channels, kernel_size_x, kernel_size_y, input_channels}), kernel_regularizer_l1, kernel_regularizer_l2 };
            auto b = variable<tensor<float»{ zeros<float>({1, 1, output_channels}), bias_regularizer_l1, bias_regularizer_l2, use_bias };
            return conv2d( input_x, input_y, stride_x, stride_y, dilation_row, dilation_col, padding )( ex, w ) + b;
        };
    }

    inline auto Conv2D( unsigned long output_channels, std::vector<unsigned long> const& kernel_size, std::string const& padding="valid",
                        std::vector<unsigned long> const& strides={1,1}, std::vector<unsigned long> const& dilations={1, 1}, bool use_bias=true,
                         float kernel_regularizer_l1=0.0f, float kernel_regularizer_l2=0.0f, float bias_regularizer_l1=0.0f, float bias_regularizer_l2=0.0f
            ) noexcept
    {

        better_assert( output_channels > 0, "Expecting output_channels larger than 0." );
        better_assert( kernel_size.size() > 0, "Expecting kernel_size at least has 1 elements." );
        better_assert( strides.size() > 0, "Expecting strides at least has 1 elements." );
        return [=]<Expression Ex>( Ex const& ex ) noexcept
        {
            unsigned long const kernel_size_x = kernel_size[0];
            unsigned long const kernel_size_y = kernel_size.size() == 2 ?  kernel_size[1] : kernel_size[0];
            //unsigned long const input_channels = input_shape[2];
            unsigned long const input_channels = *(ex.shape().rbegin());
            unsigned long const stride_x = strides[0];
            unsigned long const stride_y = strides.size() == 2 ?  strides[1] :  strides[0];
            unsigned long const dilation_row = dilations[0];
            unsigned long const dilation_col = dilations.size() == 2 ?  dilations[1] :  dilations[0];
            auto w = variable<tensor<float»{ glorot_uniform<float>({output_channels, kernel_size_x, kernel_size_y, input_channels}), kernel_regularizer_l1, kernel_regularizer_l2 };
```

```
            auto b = variable<tensor<float»{ zeros<float>({1, 1, output_channels}), bias_regularizer_l1, bias_regularizer_l2, use_bias };
            return general_conv2d( stride_x, stride_y, dilation_row, dilation_col, padding )( ex, w ) + b;
        };
    }

#if 0
    [[deprecated("input_size is not required in the new Dense()")]]
    inline auto Dense( unsigned long output_size, unsigned long input_size, bool use_bias=true, float kernel_regularizer_l1=0.0f, float kernel_regularizer_l2=0.0f, float bias_regularizer_l1=0.0f, float bias_regularizer_l2=0.0f )
    {
        return [=]<Expression Ex>( Ex const& ex )
        {
            auto w = variable<tensor<float»{ glorot_uniform<float>({input_size, output_size}), kernel_regularizer_l1, kernel_regularizer_l2 };
            auto b = variable<tensor<float»{ zeros<float>({1, output_size}), bias_regularizer_l1, bias_regularizer_l2, use_bias }; // if use_baias, then b is trainable; otherwise, non-trainable.
            return ex * w + b;
        };
    }
#endif
    inline auto Dense( unsigned long output_size, bool use_bias=true, float kernel_regularizer_l1=0.0f, float kernel_regularizer_l2=0.0f, float bias_regularizer_l1=0.0f, float bias_regularizer_l2=0.0f )
    {
        return [=]<Expression Ex>( Ex const& ex )
        {
            unsigned long const input_size = *(ex.shape().rbegin());
            auto w = variable<tensor<float>>{ glorot_uniform<float>({input_size, output_size}), kernel_regularizer_l1, kernel_regularizer_l2 };
            auto b = variable<tensor<float>>{ zeros<float>({1, output_size}), bias_regularizer_l1, bias_regularizer_l2, use_bias }; // if use_baias, then b is trainable; otherwise, non-trainable.
            return ex * w + b;
        };
    }

#if 0
    inline auto BatchNormalization( std::vector<unsigned long> const& shape, float threshold = 0.95f, float kernel_regularizer_l1=0.0f, float kernel_regularizer_l2=0.0f, float bias_regularizer_l1=0.0f, float bias_regularizer_l2=0.0f )
    {
        return [=]<Expression Ex>( Ex const& ex )
        {
            unsigned long const last_dim = *(shape.rbegin());
            auto gamma = variable{ ones<float>( {last_dim, }  ), kernel_regularizer_l1, kernel_regularizer_l2 };
            auto beta = variable{ zeros<float>( {last_dim, } ), bias_regularizer_l1, bias_regularizer_l2 };
            return batch_normalization( threshold )( ex, gamma, beta );
        };
    }

    inline auto BatchNormalization( float threshold, std::vector<unsigned long> const& shape, float kernel_regularizer_l1=0.0f, float kernel_regularizer_l2=0.0f, float bias_regularizer_l1=0.0f, float bias_regularizer_l2=0.0f )
    {
        return BatchNormalization( shape, threshold, kernel_regularizer_l1, kernel_regularizer_l2, bias_regularizer_l1, bias_regularizer_l2 );
    }
#endif
    inline auto BatchNormalization( float threshold = 0.95f, float kernel_regularizer_l1=0.0f, float kernel_regularizer_l2=0.0f, float bias_regularizer_l1=0.0f, float bias_regularizer_l2=0.0f )
    {
        return [=]<Expression Ex>( Ex const& ex )
        {
            unsigned long const last_dim = *(ex.shape().rbegin());
            auto gamma = variable{ ones<float>( {last_dim, }  ), kernel_regularizer_l1, kernel_regularizer_l2 };
            auto beta = variable{ zeros<float>( {last_dim, } ), bias_regularizer_l1, bias_regularizer_l2 };
            return batch_normalization( threshold )( ex, gamma, beta );
        };
    }

#if 0
    // TODO: fix this layer
    inline auto LayerNormalization( std::vector<unsigned long> const& shape )
    {
        return [=]<Expression Ex>( Ex const& ex )
        {
            unsigned long const last_dim = *(shape.rbegin());
            auto gamma = variable<tensor<float»{ ones<float>( {last_dim, }  ) };
            auto beta = variable<tensor<float»{ zeros<float>( {last_dim, } ) };
            return layer_normalization()( ex, gamma, beta );
        };
    }
```

```
#endif

    inline auto Concatenate(unsigned long axis = -1) noexcept
    {
        return [=]<Expression Lhs_Expression, Expression Rhs_Expression>( Lhs_Expression const& lhs_ex, Rhs_Expression const& rhs_ex ) noexcept
        {
            return concatenate( axis )( lhs_ex, rhs_ex );
        };
    }

    inline auto Add() noexcept
    {
        return []<Expression Lhs_Expression, Expression Rhs_Expression>( Lhs_Expression const& lhs_ex, Rhs_Expression const& rhs_ex ) noexcept
        {
            return lhs_ex + rhs_ex;
        };
    }


    inline auto Subtract() noexcept
    {
        return []<Expression Lhs_Expression, Expression Rhs_Expression>( Lhs_Expression const& lhs_ex, Rhs_Expression const& rhs_ex ) noexcept
        {
            return lhs_ex - rhs_ex;
        };
    }

    inline auto Multiply() noexcept
    {
        return []<Expression Lhs_Expression, Expression Rhs_Expression>( Lhs_Expression const& lhs_ex, Rhs_Expression const& rhs_ex ) noexcept
        {
            return hadamard_product( lhs_ex, rhs_ex );
        };
    }

    template< Expression Ex >
    inline auto ReLU( Ex const& ex ) noexcept
    {
        return relu( ex );
    }

    inline auto Softmax() noexcept
    {
        return []< Expression Ex >( Ex const& ex ) noexcept
        {
            return softmax( ex );
        };
    }


    template< typename T = float >
    inline auto LeakyReLU( T const factor=0.2 ) noexcept
    {
        return leaky_relu( factor );
    }

    template< typename T = float >
    inline auto ELU( T const factor=0.2 ) noexcept
    {
        return elu( factor );
    }


    inline auto Reshape( std::vector<unsigned long> const& new_shape, bool include_batch_flag=true ) noexcept
    {
        return reshape( new_shape, include_batch_flag );
    }

    inline auto Flatten() noexcept
    {
        return []<Expression Ex>( Ex const& ex ) noexcept
        {
            return flatten( ex );
        };
    }

    inline auto MaxPooling2D( unsigned long stride ) noexcept
    {
        return max_pooling_2d( stride );
    }

    inline auto UpSampling2D( unsigned long stride ) noexcept
```

```
    {
        return up_sampling_2d( stride );
    }

    template< typename T >
    inline auto Dropout( T factor ) noexcept
    {
        return drop_out( factor );
    }

    inline auto AveragePooling2D( unsigned long stride ) noexcept
    {
        return average_pooling_2d( stride );
    }

    //
    // TODO: wrap more operations from 'operation.hpp'
    //








}//namespace f

#endif//NLESIGQPSASUTOXPLGXCUHFGGUGYSWLQQFATNISJOSPUFHRORXBNXLSWTYRNSIWJKYFXIQXVN

```

9.17 /home/feng/workspace/github.repo/ceras/include/loss.hpp File Reference

```
#include "./operation.hpp"
#include "./tensor.hpp"
#include "./utils/debug.hpp"
```

Namespaces

- namespace ceras

Functions

- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto ceras::mean_squared_logarithmic_error (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto ceras::squared_loss (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto ceras::mean_squared_error (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto ceras::mse (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto ceras::abs_loss (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto ceras::mean_absolute_error (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept

- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto ceras::mae (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto ceras::cross_entropy (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto ceras::binary_cross_entropy_loss (Lhs_Expression const &ground_truth, Rhs_Expression const &prediction) noexcept
- template<Expression Lhs_Expression, Expression Rhs_Expression, std::floating_point F = float>
  constexpr auto ceras::cross_entropy_loss (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex, F label_smoothing_factor=0.0) noexcept
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto ceras::hinge_loss (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept

Variables

- static constexpr auto ceras::MeanSquaredError

  Computes the mean of squares of errors between labels and predictions.
- static constexpr auto ceras::MSE

  An alias name of function MeanSquaredError.
- static constexpr auto ceras::MeanAbsoluteError

  Computes the mean of absolute errors between labels and predictions.
- static constexpr auto ceras::MAE

  An alias name of function MeanAbsoluteError.
- static constexpr auto ceras::Hinge
- static constexpr auto ceras::CategoricalCrossentropy
- static constexpr auto ceras::CategoricalCrossEntropy
- static constexpr auto ceras::BinaryCrossentropy
- static constexpr auto ceras::BinaryCrossEntropy

9.18 loss.hpp

Go to the documentation of this file.

```
#ifndef APWVIJWMXHAVXUGYGVNDSEFKTMBKLBMGLSHWUPRPGLFCHUBDRAHGSTDSEDNKOGTIBNQVNLXCD
#define APWVIJWMXHAVXUGYGVNDSEFKTMBKLBMGLSHWUPRPGLFCHUBDRAHGSTDSEDNKOGTIBNQVNLXCD

#include "./operation.hpp"
#include "./tensor.hpp"
#include "./utils/debug.hpp"

namespace ceras
{

     template < Expression Lhs_Expression, Expression Rhs_Expression >
     auto constexpr mean_squared_logarithmic_error( Lhs_Expression const& lhs_ex, Rhs_Expression const& rhs_ex ) noexcept
     {
         return sum_reduce( square( minus(log( value{1.0} + clip(eps)(lhs_ex) ), log( value{1.0} + clip(eps)(rhs_ex) ))) );
     }

     template < Expression Lhs_Expression, Expression Rhs_Expression >
     auto constexpr squared_loss( Lhs_Expression const& lhs_ex, Rhs_Expression const& rhs_ex ) noexcept
     {
         return sum_reduce( square( minus(lhs_ex, rhs_ex)) );
     }

     template < Expression Lhs_Expression, Expression Rhs_Expression >
     auto constexpr mean_squared_error( Lhs_Expression const& lhs_ex, Rhs_Expression const& rhs_ex ) noexcept
     {
         return mean_reduce( square( minus(lhs_ex, rhs_ex)) );
```

```
}

template < Expression Lhs_Expression, Expression Rhs_Expression >
auto constexpr mse( Lhs_Expression const& lhs_ex, Rhs_Expression const& rhs_ex ) noexcept
{
    return mean_squared_error( lhs_ex, rhs_ex );
}

template < Expression Lhs_Expression, Expression Rhs_Expression >
auto constexpr abs_loss( Lhs_Expression const& lhs_ex, Rhs_Expression const& rhs_ex ) noexcept
{
    return sum_reduce( abs( minus(lhs_ex, rhs_ex)) );
}

template < Expression Lhs_Expression, Expression Rhs_Expression >
auto constexpr mean_absolute_error( Lhs_Expression const& lhs_ex, Rhs_Expression const& rhs_ex ) noexcept
{
    return mean_reduce( abs( minus(lhs_ex, rhs_ex)) );
};

template < Expression Lhs_Expression, Expression Rhs_Expression >
auto constexpr mae( Lhs_Expression const& lhs_ex, Rhs_Expression const& rhs_ex ) noexcept
{
    return mean_absolute_error( lhs_ex, rhs_ex );
};

template < Expression Lhs_Expression, Expression Rhs_Expression >
auto constexpr cross_entropy( Lhs_Expression const& lhs_ex, Rhs_Expression const& rhs_ex ) noexcept
{
    return negative( sum_reduce( hadamard_product( lhs_ex, log(rhs_ex) ) ) );
}

namespace
{
    struct cross_entropy_loss_context
    {
        template< std::floating_point T >
        auto make_forward( T label_smoothing_factor ) const noexcept
        {
            return [label_smoothing_factor]<Tensor Tsor>( Tsor const& ground_truth_input, Tsor const& prediction_input ) noexcept
            {
               Tsor sm = softmax( prediction_input );
               typedef typename Tsor::value_type value_type;
               typename Tsor::value_type ans{0};
               unsigned long const n = *(ground_truth_input.shape().rbegin());
               value_type const _c0 = label_smoothing_factor / (n-1);
               value_type const _c1 = value_type{1} - label_smoothing_factor;

               for ( auto idx :  range( ground_truth_input.size() ) )
               {
                   value_type const v = ground_truth_input[idx] > eps ?  _c1 :  _c0;
                   ans -= v * std::log( std::max( static_cast<typename Tsor::value_type>(eps), sm[idx] ) );
                   //ans -= ground_truth_input[idx] * std::log( std::max( static_cast<typename Tsor::value_type>(eps), sm[idx] ) );
               }
               auto result = as_tensor<typename Tsor::value_type, typename Tsor::allocator>(ans/(*(ground_truth_input.shape().begin())));
               return result;
            };
        }

        template< std::floating_point T >
        auto make_backward( T label_smoothing_factor) const noexcept
        {
            return [=]<Tensor Tsor>( Tsor const& ground_truth_input, Tsor const& prediction_input, [[maybe_unused]]Tsor const& output_data, [[maybe_unused]]Tsor const& grad ) noexcept
            {
               // In our implementation, the grad is always 1, unless this layer is nested contributing to a combined weighted loss
               typedef typename Tsor::value_type value_type;
               value_type const factor = grad[0]; // the shape of grad is {1,}

               unsigned long const n = *(ground_truth_input.shape().rbegin());
               value_type const _c0 = label_smoothing_factor / (n-1);
               value_type const _c1 = value_type{1} - label_smoothing_factor;

               Tsor ground_truth_gradient = ground_truth_input;

                //Tsor sm = softmax( prediction_input ) - ground_truth_input;
                //return std::make_tuple( ground_truth_gradient*factor, sm*factor );

                Tsor sm = softmax( prediction_input );
                for ( auto idx :  range( ground_truth_input.size() ) )
                {
```

```
                     value_type const v = ground_truth_gradient[idx] > eps ?  _c1 :  _c0;
                     sm[idx] = factor * (sm[idx] - v );
                 }

                 return std::make_tuple( ground_truth_gradient*factor, sm );
              };
          }

      };//struct cross_entropy_loss_context

  }//anonymous namespace

  template < Expression Lhs_Expression, Expression Rhs_Expression >
  auto constexpr binary_cross_entropy_loss( Lhs_Expression const& ground_truth, Rhs_Expression const& prediction ) noexcept
  {
      auto ones = ones_like( ground_truth );
      auto error = negative( hadamard_product( ground_truth, log(prediction) ) + hadamard_product( (ones - ground_truth), log(ones - prediction) ) );
      return mean_reduce( error );
  }


  // beware:  do not apply softmax activation before this layer, as this loss is softmax+xentropy already
  template < Expression Lhs_Expression, Expression Rhs_Expression, std::floating_point F=float >
  auto constexpr cross_entropy_loss( Lhs_Expression const& lhs_ex, Rhs_Expression const& rhs_ex, F label_smoothing_factor=0.0 ) noexcept
  {
      return make_binary_operator( cross_entropy_loss_context{}.make_forward( label_smoothing_factor ), cross_entropy_loss_context{}.make_backward( label_smoothing_factor ), "CrossEntropyLoss" )( lhs_ex, rhs_ex );
  }

  template < Expression Lhs_Expression, Expression Rhs_Expression >
  auto constexpr hinge_loss( Lhs_Expression const& lhs_ex, Rhs_Expression const& rhs_ex ) noexcept
  {
      return mean_reduce( maximum( value{0.0f}, value{1.0f} - hadamard_product(lhs_ex, rhs_ex) ) );
  }

  // loss interfaces
  // A loss is an expression.  This expression takes two parameters.
  // The first parameter is a place_holder, that will be binded to an tensor.
  // The second parameter is an expression, that will be evaluated to compare with the tensor binded to the first parameter

  inline static constexpr auto MeanSquaredError = []()
  {
      return []<Expression Ex >( Ex const& output )
      {
          return [=]<Place_Holder Ph>( Ph const& ground_truth )
          {
              return mean_squared_error( ground_truth, output );
          };
      };
  };

  inline static constexpr auto MSE = []()
  {
      return MeanSquaredError();
  };

  inline static constexpr auto MeanAbsoluteError = []()
  {
      return []<Expression Ex >( Ex const& output )
      {
          return [=]<Place_Holder Ph>( Ph const& ground_truth )
          {
              return mean_absolute_error( ground_truth, output );
          };
      };
  };


  inline static constexpr auto MAE = []()
  {
      return MeanAbsoluteError();
  };



  inline static constexpr auto Hinge = []()
  {
      return []<Expression Ex >( Ex const& output )
      {
          return [=]<Place_Holder Ph>( Ph const& ground_truth )
          {
```

```
                return hinge_loss( ground_truth, output );
            };
        };
    };

    // note:  do not apply softmax activation to the last layer of the model, this loss has packaged it
    inline static constexpr auto CategoricalCrossentropy = []<std::floating_point F=float>( F label_smoothing_factor = 0.0)
    {
        return [=]<Expression Ex >( Ex const& output )
        {
            return [=]<Place_Holder Ph>( Ph const& ground_truth )
            {
                return cross_entropy_loss( ground_truth, output, label_smoothing_factor );
            };
        };
    };

    inline static constexpr auto CategoricalCrossEntropy = []<std::floating_point F=float>(F label_smoothing_factor = 0.0)
    {
        return CategoricalCrossentropy(label_smoothing_factor);
    };

    inline static constexpr auto BinaryCrossentropy = []()
    {
        return []<Expression Ex >( Ex const& output )
        {
            return [=]<Place_Holder Ph>( Ph const& ground_truth )
            {
                return binary_cross_entropy_loss( ground_truth, output );
            };
        };
    };

    inline static constexpr auto BinaryCrossEntropy = []()
    {
        return BinaryCrossentropy();
    };


}//namespace ceras

#endif//APWVIJWMXHAVXUGYGVNDSEFKTMBKLBMGLSHWUPRPGLFCHUBDRAHGSTDSEDNKOGTIBNQVNLXCD

```

9.19 /home/feng/workspace/github.repo/ceras/include/metric.hpp File Reference

```
#include "./operation.hpp"
#include "./activation.hpp"
#include "./loss.hpp"
```

Namespaces

- namespace ceras

Functions

- template<Expression Lhs_Expression, Expression Rhs_Expression, std::floating_point FP>
  auto ceras::binary_accuracy (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex, FP threshold=0.5) noexcept

9.20 metric.hpp

Go to the documentation of this file.

```
#ifndef EIDQGGJMELOKMYSJNSKIUQTTOGNSMKBVITQSMTSVJDNWXPJBJKQBJDAIAIYMTPTIBLWPBCYLI
#define EIDQGGJMELOKMYSJNSKIUQTTOGNSMKBVITQSMTSVJDNWXPJBJKQBJDAIAIYMTPTIBLWPBCYLI

#include "./operation.hpp"
#include "./activation.hpp"
#include "./loss.hpp"

namespace ceras
{

    template< Expression Lhs_Expression, Expression Rhs_Expression, std::floating_point FP >
    auto binary_accuracy( Lhs_Expression const& lhs_ex, Rhs_Expression const& rhs_ex, FP threshold=0.5 )
    noexcept
    {
        return mean( equal( lhs_ex, rhs_ex ) );
    }



}//namespace ceras

#endif//EIDQGGJMELOKMYSJNSKIUQTTOGNSMKBVITQSMTSVJDNWXPJBJKQBJDAIAIYMTPTIBLWPBCYLI

```

9.21 /home/feng/workspace/github.repo/ceras/include/model.hpp File Reference

```
#include "./includes.hpp"
#include "./operation.hpp"
#include "./place_holder.hpp"
#include "./tensor.hpp"
#include "./utils/better_assert.hpp"
#include "./utils/context_cast.hpp"
#include "./utils/tqdm.hpp"
```

Classes

- struct ceras::compiled_model< Model, Optimizer, Loss >
- struct ceras::model< Ex, Ph >

Namespaces

- namespace ceras

Functions

- template<Expression Ex>
  void ceras::make_trainable (Ex &ex, bool t)
- template<Expression Ex, Place_Holder Ph, Expression Ey>
  auto ceras::replace_placeholder_with_expression (Ex const &ex, Ph const &old_place_holder, Ey const &new_expression)
- template<typename Model , typename Optimizer , typename Loss >
  auto ceras::make_compiled_model (Model const &m, Loss const &l, Optimizer const &o)

9.22 model.hpp

Go to the documentation of this file.

```cpp
#ifndef BPLYFMIFNNWSGMLLEKBJMAJDBRSPHHRAYMOHTWSTCMNMFSLLYNQTTCCAQXKXSLMSLKESHRASL
#define BPLYFMIFNNWSGMLLEKBJMAJDBRSPHHRAYMOHTWSTCMNMFSLLYNQTTCCAQXKXSLMSLKESHRASL

#include "./includes.hpp"
#include "./operation.hpp"
#include "./place_holder.hpp"
#include "./tensor.hpp"
#include "./utils/better_assert.hpp"
#include "./utils/context_cast.hpp"
#include "./utils/tqdm.hpp"

namespace ceras
{
    template< Expression Ex >
    void make_trainable( Ex& ex , bool t )
    {
        if constexpr (is_variable_v<Ex>)
        {
            ex.trainable( t );
        }
        else if constexpr (is_binary_operator_v<Ex>)
        {
            make_trainable( ex.lhs_op_, t );
            make_trainable( ex.rhs_op_, t );
        }
        else if constexpr (is_unary_operator_v<Ex>)
        {
            make_trainable( ex.op_, t );
        }
    }

    template< Expression Ex, Place_Holder Ph, Expression Ey >
    auto replace_placeholder_with_expression( Ex const& ex, Ph const& old_place_holder, Ey const& new_expression )
    {
        if constexpr (is_value_v<Ex> || is_constant_v<Ex> || is_variable_v<Ex>)
        {
            return ex;
        }
        else if constexpr (is_place_holder_v<Ex>)
        {
            return new_expression; // assuming only one place holder in the model
            //return (ex == old_place_holder) ?  new_expression :  ex;
        }
        else if constexpr (is_unary_operator_v<Ex>)
        {
            return make_unary_operator( ex.forward_action_, ex.backward_action_, ex.name_, ex.output_shape_calculator_ )( replace_placeholder_with_expression( ex.op_, old_place_holder, new_expression ) );
        }
        else if constexpr (is_binary_operator_v<Ex>)
        {
            return make_binary_operator( ex.forward_action_, ex.backward_action_, ex.name_, ex.output_shape_calculator_ )
                                       ( replace_placeholder_with_expression( ex.lhs_op_, old_place_holder, new_expression ),
                                         replace_placeholder_with_expression( ex.rhs_op_, old_place_holder, new_expression ) );
        }
        else
        {
            better_assert( false, "replace::Should never reach here!" );
        }
    }

    template< typename Model, typename Optimizer, typename Loss >
    struct compiled_model
    {
        typedef typename Model::input_layer_type io_layer_type;
        typedef decltype(std::declval<Optimizer>()(std::declval<Loss&>())) optimizer_type; // defined because the compiled optimizer takes a reference to an expression as its parameter

        Model model_;
        io_layer_type input_place_holder_;
        io_layer_type ground_truth_place_holder_;
        Loss loss_; // MeanSquaredError()( model_.output() )( find a input );
        Optimizer optimizer_; // Adam( ...  )
        optimizer_type compiled_optimizer_;

        compiled_model( Model const& m, io_layer_type const& input_place_holder, io_layer_type const& ground_truth_place_holder, Loss const& loss, Optimizer const& optimizer ):
```

```
         model_{m}, input_place_holder_{input_place_holder}, ground_truth_place_holder_{ground_truth_place_holder}, loss_{loss}, optimizer_{optimizer}, compiled_optimizer_{ optimizer_(loss_) }  { }

     template< Tensor Tsor >
     auto evaluate( Tsor const& inputs, Tsor const& outputs, unsigned long batch_size=32 )
     {
         // extract size of samples
         unsigned long const samples = *(inputs.shape().begin());
         unsigned long const loops = samples / batch_size;

          // prepare tensor for inputs
          std::vector<unsigned long> batch_input_shape = inputs.shape();
          batch_input_shape[0] = batch_size;
          Tsor input_samples{ batch_input_shape };
          unsigned long const input_size_per_batch = input_samples.size();

          // prepare tensor for outputs
          std::vector<unsigned long> batch_output_shape = outputs.shape();
          batch_output_shape[0] = batch_size;
          Tsor output_samples{ batch_output_shape };
          unsigned long const output_size_per_batch = output_samples.size();

          // bind tensors to place holders
          //session<Tsor> s;
          auto& s = get_default_session<Tsor>();
          s.bind( input_place_holder_, input_samples );
          s.bind( ground_truth_place_holder_, output_samples );

          typedef typename Tsor::value_type value_type;
          value_type validation_error = 0;

          learning_phase = 0; // for different behaviours in normalization and drop-out layers

          for ( auto l :  tq::trange( loops ) )
          {
              // feed data
              std::copy_n( inputs.data() + l * input_size_per_batch, input_size_per_batch, input_samples.data() );
              std::copy_n( outputs.data() + l * output_size_per_batch, output_size_per_batch, output_samples.data() );
              // forward pass
              auto error = s.run( loss_ ).as_scalar();
              // in case of training split, do backpropagation
              validation_error += error;
          }

          learning_phase = 1; // for different behaviours in normalization and drop-out layers

          return validation_error / loops;
      }

      template< Tensor Tsor >
      auto fit( Tsor const& inputs, Tsor const& outputs, unsigned long batch_size, unsigned long epoch=1, int verbose=0, double validation_split=0.0 )
      {
          // extract size of samples
          unsigned long const samples = *(inputs.shape().begin());
          unsigned long const loops_per_epoch = samples / batch_size;
          unsigned long const training_loops = ( 1.0 - validation_split ) * loops_per_epoch;
          unsigned long const validation_loops = loops_per_epoch - training_loops;

          // prepare tensor for inputs
          std::vector<unsigned long> batch_input_shape = inputs.shape();
          batch_input_shape[0] = batch_size;
          Tsor input_samples{ batch_input_shape };
          unsigned long const input_size_per_batch = input_samples.size();

          // prepare tensor for outputs
          std::vector<unsigned long> batch_output_shape = outputs.shape();
          batch_output_shape[0] = batch_size;
          Tsor output_samples{ batch_output_shape };
          unsigned long const output_size_per_batch = output_samples.size();

          // bind tensors to place holders
          //session<Tsor> s;
          auto& s = get_default_session<Tsor>();//.get();
          s.bind( input_place_holder_, input_samples );
          s.bind( ground_truth_place_holder_, output_samples );

          // collect training errors
          typedef typename Tsor::value_type value_type;
          std::vector<value_type> training_errors;
          std::vector<value_type> validation_errors;

          learning_phase = 1; // for different behaviours in normalization and drop-out layers

```

```
            for ( auto e :  range( epoch ) )
            {
                value_type training_error = 0;
                value_type validation_error = 0;
                for ( auto l :  tq::trange( loops_per_epoch ) )
                {
                    // feed data
                    std::copy_n( inputs.data() + l * input_size_per_batch, input_size_per_batch, input_samples.data() );
                    std::copy_n( outputs.data() + l * output_size_per_batch, output_size_per_batch, output_samples.data() );
                    // forward pass
                    auto error = s.run( loss_ ).as_scalar();
                    // in case of training split, do backpropagation
                    if ( l <= training_loops )
                    {
                        training_error += error;
                        s.run( compiled_optimizer_ );
                    }
                    else // in case of validation split, just collect errors
                    {
                        validation_error += error;
                    }
                }
                training_errors.push_back( training_error / training_loops );
                validation_errors.push_back( validation_error / validation_loops );
                if ( verbose )
                    std::cout « "\nTraining error:  " « training_error / training_loops « " and validation error:  " « validation_error / validation_loops « " at epoch:  " « e+1 « "/" « epoch;
                std::cout « std::endl;
            }
            return std::make_tuple( training_errors, validation_errors );
        }

        template< Tensor Tsor >
        auto train_on_batch( Tsor const& input, Tsor const& output )
        {
            learning_phase = 1; // for different behaviours in normalization and drop-out layers
            auto& s = get_default_session<Tsor>();//.get();
            s.bind( input_place_holder_, input );
            s.bind( ground_truth_place_holder_, output );
            //debug_log( "Training on batch forward pass." );
            auto error = s.run( loss_ );
            //debug_log( "Training on batch backward pass." );
            s.run( compiled_optimizer_ );
            return error.as_scalar();
        }

        template< Tensor Tsor>
        auto predict( Tsor const& input_tensor )
        {
            auto m = model_;
            return m.predict( input_tensor );
        }

        template< Expression Exp >
        auto operator()( Exp const& ex ) const noexcept
        {
            return model_( ex );
        }

        void trainable( bool t )
        {
            model_.trainable( t );
        }
    };

    template< typename Model, typename Optimizer, typename Loss >
    inline auto make_compiled_model( Model const& m, Loss const& l, Optimizer const& o )
    {
        auto input_place_holder = m.input();
        auto ground_truth_place_holder = typename Model::input_layer_type{};
        auto loss = l( m.output() )( ground_truth_place_holder );
        auto optimizer = o( loss );
        return compiled_model{ m, input_place_holder, ground_truth_place_holder, loss, o};
    }

    template< Expression Ex, Place_Holder Ph >
    struct model
    {
        typedef Ph       input_layer_type;
        typedef Ex       output_layer_type;

        output_layer_type expression_;
        input_layer_type place_holder_;//< input layer of the model.


```

```
        input_layer_type input() const noexcept { return place_holder_; }

        output_layer_type output() const noexcept { return expression_; }

        model( input_layer_type const& place_holder, output_layer_type const& expression ) :
    expression_{expression}, place_holder_{place_holder} {}

        template< Tensor Tsor>
        auto predict( Tsor const& input_tensor )
        {
            learning_phase = 0; // for different behaviours in normalization and drop-out layers

            //session<Tsor> s;
            auto& s = get_default_session<Tsor>();//.get();
            s.bind( place_holder_, input_tensor );

            auto ans = s.run( expression_ );

            learning_phase = 1; // restore learning phase

            return ans;
        }

        template< Expression Exp >
        auto operator()( Exp const& ex ) const noexcept
        {
            return replace_placeholder_with_expression( expression_, place_holder_, ex );
        }

        template< typename Loss, typename Optimizer >
        auto compile( Loss const& l, Optimizer const& o )
        {
            return make_compiled_model( *this, l, o );
        }

        void trainable( bool t )
        {
            make_trainable( expression_, t );
        }


        void save_weights( std::string const& file )
        {
            auto& s = get_default_session<tensor<float»();
            s.serialize( file );
        }

        void load_weights( std::string const& file )
        {
            auto& s = get_default_session<tensor<float»();
            s.deserialize( file );
        }


        void summary(std::string const& file_name=std::string{}) const noexcept
        {
            auto g = computation_graph( expression_ );

            if ( file_name.empty() )
            {
                std::cout « g « std::endl;
                return;
            }

            std::ofstream ofs{ file_name };
            ofs « g;
        }


    };

}//namespace ceras

#endif//BPLYFMIFNNWSGMLLEKBJMAJDBRSPHHRAYMOHTWSTCMNMFSLLYNQTTCCAQXKXSLMSLKESHRASLCalculate the loss for
    the model in test model
```

9.23 /home/feng/workspace/github.repo/ceras/include/operation.hpp File Reference

```
#include "./includes.hpp"
#include "./place_holder.hpp"
```

```
#include "./variable.hpp"
#include "./constant.hpp"
#include "./value.hpp"
#include "./utils/range.hpp"
#include "./utils/debug.hpp"
#include "./config.hpp"
#include "./utils/context_cast.hpp"
#include "./utils/for_each.hpp"
#include "./utils/id.hpp"
#include "./utils/enable_shared.hpp"
#include "./utils/fmt.hpp"
```

Classes

- struct ceras::identity_output_shape_calculator

  The default identity output shape calculator for unary/binary operators. Should be overrided for some special operators.
- struct ceras::unary_operator< Operator, Forward_Action, Backward_Action, Output_Shape_Calculator >

  A unary operator is composed of a.) an input expression, b.) a forward action and c.) a backward action.
- struct ceras::binary_operator< Lhs_Operator, Rhs_Operator, Forward_Action, Backward_Action, Output_Shape_Calculator >

  A binary operator is composed of a.) a left-side input expression, b.) a right-side input expression, c.) a forward action and d.) a backward action.
- struct ceras::is_unary_operator< T >
- struct ceras::is_unary_operator< unary_operator< Operator, Forward_Action, Backward_Action, Output_Shape_Calculator >
- struct ceras::is_binary_operator< T >
- struct ceras::is_binary_operator< binary_operator< Lhs_Operator, Rhs_Operator, Forward_Action, Backward_Action, Output_

Namespaces

- namespace ceras

Concepts

- concept ceras::Unary_Operator

  A type that represents an unary operator.
- concept ceras::Binary_Operator

  A type that represents a binary operator.
- concept ceras::Operator

  A type that represents an unary or a binary operator.
- concept ceras::Expression

  A type that represents a unary operator, a binary operator, a variable, a place_holder, a constant or a value.

Functions

- template<typename Forward_Action , typename Backward_Action , typename Output_Shape_Calculator = identity_output_shape_calculator>
  constexpr auto ceras::make_unary_operator (Forward_Action const &unary_forward_action, Backward_Action const &unary_backward_action, std::string const &name="Anonymous Unary Operator", Output_Shape_Calculator const &output_shape_calculator=Output_Shape_Calculator{}) noexcept

  Construct an unary operator by passing the forward/backward actions and output shape calculator.
- template<typename Forward_Action , typename Backward_Action , typename Output_Shape_Calculator = identity_output_shape_calculator>
  auto ceras::make_binary_operator (Forward_Action const &binary_forward_action, Backward_Action const &binary_backward_action, std::string const &name="Anonymous Binary Operator", Output_Shape_Calculator const &output_shape_calculator=Output_Shape_Calculator{}) noexcept
- template<Expression Ex>
  std::string ceras::computation_graph (Ex const &ex) noexcept
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto ceras::plus (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto ceras::operator+ (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept
- template<Expression Ex>
  constexpr auto ceras::operator+ (Ex const &ex) noexcept
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  auto ceras::operator* (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept
- template<Expression Ex>
  constexpr auto ceras::negative (Ex const &ex) noexcept

  Negative operator, elementwise.
- template<Expression Ex>
  constexpr auto ceras::operator- (Ex const &ex) noexcept
- template<Expression Ex>
  constexpr auto ceras::inverse (Ex const &ex) noexcept

  Inverse operator, elementwise.
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto ceras::elementwise_product (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept

  Multiply two input operators, elementwise.
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto ceras::elementwise_multiply (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto ceras::hadamard_product (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto ceras::divide (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept

  Divide one tensor by the other.
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto ceras::operator/ (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept

  Divide one tensor by the other.
- template<Expression Ex>
  constexpr auto ceras::sum_reduce (Ex const &ex) noexcept

  Sum up all elements, returns a scalar.
- template<Expression Ex>
  constexpr auto ceras::reduce_sum (Ex const &ex) noexcept
- template<Expression Ex>
  constexpr auto ceras::mean_reduce (Ex const &ex) noexcept

  Computes the mean of elements across all dimensions of an expression.

- template<Expression Ex>
  constexpr auto ceras::reduce_mean (Ex const &ex) noexcept

  An alias name of mean_reduce.
- template<Expression Ex>
  constexpr auto ceras::mean (Ex const &ex) noexcept

  An alias name of mean_reduce.
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto ceras::minus (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto ceras::operator- (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept
- template<Expression Ex>
  constexpr auto ceras::square (Ex const &ex) noexcept
- template<Expression Ex, Expression Ey>
  ∗endcode ∗∗constexpr auto hypot (Ex const &ex, Ey const &ey) noexcept
- template<typename Float >
  requires std::floating_point<Float>
  constexpr auto clip (Float lower, Float upper=std::numeric_limits< Float >::max()) noexcept
- auto reshape (std::vector< unsigned long > const &new_shape, bool include_batch_flag=true) noexcept
- template<Expression Ex>
  constexpr auto flatten (Ex const &ex) noexcept

  Flatten input tensor.
- constexpr auto expand_dims (int axis=-1) noexcept

  Expand input tensor with a length 1 axis inserted at index axis.
- auto argmax (unsigned long axis=0) noexcept

  Returns the index with the largest value across axes of an input tensor.
- auto argmin (unsigned long axis=0) noexcept

  Returns the index with the smallest value across axes of an input tensor.
- template<Expression Ex>
  constexpr auto identity (Ex const &ex) noexcept

  Identity operation.
- template<Expression Ex>
  auto transpose (Ex const &ex) noexcept

  Transpose a matrix.
- auto img2col (unsigned long const row_kernel, unsigned long col_kernel=-1, unsigned long const row_↩ padding=0, unsigned long col_padding=0, unsigned long const row_stride=1, unsigned long const col_↩ stride=1, unsigned long const row_dilation=1, unsigned long const col_dilation=1) noexcept
- auto conv2d (unsigned long row_input, unsigned long col_input, unsigned long const row_stride=1, unsigned long const col_stride=1, unsigned long const row_dilation=1, unsigned long const col_dilation=1, std::string const &padding="valid") noexcept
- auto general_conv2d (unsigned long const row_stride=1, unsigned long const col_stride=1, unsigned long const row_dilation=1, unsigned long const col_dilation=1, std::string const &padding="valid") noexcept

  Conv2D not constrained by the input shape.
- template<typename T >
  requires std::floating_point<T>
  auto drop_out (T const factor) noexcept
- template<typename T >
  requires std::floating_point<T>
  auto dropout (T const factor) noexcept

  dropout is an alias name of drop_out.
- auto max_pooling_2d (unsigned long stride) noexcept
- auto average_pooling_2d (unsigned long stride) noexcept
- auto up_sampling_2d (unsigned long stride) noexcept
- auto upsampling_2d (unsigned long stride) noexcept

- template<typename T = double>
  requires std::floating_point<T>
  auto normalization_batch (T const momentum=0.98) noexcept
- template<typename T >
  requires std::floating_point<T>
  auto batch_normalization (T const momentum=0.98) noexcept
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto concatenate (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept
- auto concatenate (unsigned long axe=-1)
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto concat (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept
- auto concat (unsigned long axe=-1)
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto maximum (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto minimum (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept
- template<Expression Lhs_Expression, Expression Rhs_Expression>
  constexpr auto atan2 (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept

  Computes the arc tangent of y/x using the signs of arguments to determine the correct quadrant.
- template<typename T = float>
  requires std::floating_point<T>
  auto random_normal_like (T mean=0.0, T stddev=1.0) noexcept
- template<Expression Ex>
  auto ones_like (Ex const &ex) noexcept
- template<Expression Ex>
  auto zeros_like (Ex const &ex) noexcept
- template<Expression Lhs_Expression, Expression Rhs_Expression, std::floating_point FP>
  constexpr auto equal (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex, FP threshold=0.5) noexcept
- template<Expression Ex>
  constexpr auto sign (Ex const &ex) noexcept
- auto zero_padding_2d (std::vector< unsigned long > const &padding) noexcept

  Zero-padding layer for 2D input. The input should have 4-dimensions: `(batch_size, row, col, channel)`*. The output has 4-dimensions:* `(batch_size, new_row, new_col, channel)`.
- auto cropping_2d (std::vector< unsigned long > const &padding) noexcept

  Cropping layer for 2D input. The input should have 4-dimensions: `(batch_size, row, col, channel)`*. The output has 4-dimensions:* `(batch_size, new_row, new_col, channel)`.
- auto sliding_2d (unsigned long pixels) noexcept
- auto repeat (unsigned long repeats, unsigned long axis=-1) noexcept

  Repeats elements along an axis.
- auto reduce_min (unsigned long axis=-1) noexcept

  Reduce minimal elements along an axis.
- auto reduce_max (unsigned long axis=-1) noexcept

  Reduce maximum elements along an axis.
- auto reduce_sum (unsigned long axis) noexcept

  Reduce sum elements along an axis.
- template<Expression Ex>
  constexpr auto abs (Ex const &ex) noexcept

  Computes Abs of the given expression.
- template<Expression Ex>
  constexpr auto acos (Ex const &ex) noexcept

  Computes Acos of the given expression.
- template<Expression Ex>
  constexpr auto acosh (Ex const &ex) noexcept

Computes Acosh of the given expression.

- template<Expression Ex>
  constexpr auto asin (Ex const &ex) noexcept

  Computes Asin of the given expression.
- template<Expression Ex>
  constexpr auto asinh (Ex const &ex) noexcept

  Computes Asinh of the given expression.
- template<Expression Ex>
  constexpr auto atan (Ex const &ex) noexcept

  Computes Atan of the given expression.
- template<Expression Ex>
  constexpr auto atanh (Ex const &ex) noexcept

  Computes Atanh of the given expression.
- template<Expression Ex>
  constexpr auto cbrt (Ex const &ex) noexcept

  Computes Cbert of the given expression.
- template<Expression Ex>
  constexpr auto ceil (Ex const &ex) noexcept

  Computes Ceil of the given expression.
- template<Expression Ex>
  constexpr auto cos (Ex const &ex) noexcept

  Computes Cos of the given expression.
- template<Expression Ex>
  constexpr auto cosh (Ex const &ex) noexcept

  Computes Cosh of the given expression.
- template<Expression Ex>
  constexpr auto erf (Ex const &ex) noexcept

  Computes Erf of the given expression.
- template<Expression Ex>
  constexpr auto erfc (Ex const &ex) noexcept

  Computes Erfc of the given expression.
- template<Expression Ex>
  constexpr auto exp (Ex const &ex) noexcept

  Computes Exp of the given expression.
- template<Expression Ex>
  constexpr auto exp2 (Ex const &ex) noexcept

  Computes Exp2 of the given expression.
- template<Expression Ex>
  constexpr auto expm1 (Ex const &ex) noexcept

  Computes Expm1 of the given expression.
- template<Expression Ex>
  constexpr auto fabs (Ex const &ex) noexcept

  Computes Fabs of the given expression.
- template<Expression Ex>
  constexpr auto floor (Ex const &ex) noexcept

  Computes Floor of the given expression.
- template<Expression Ex>
  constexpr auto llrint (Ex const &ex) noexcept

  Computes Llrint of the given expression.
- template<Expression Ex>
  constexpr auto llround (Ex const &ex) noexcept

  Computes Llround of the given expression.

- template<Expression Ex>
  constexpr auto log (Ex const &ex) noexcept

  Computes Log of the given expression.
- template<Expression Ex>
  constexpr auto log10 (Ex const &ex) noexcept

  Computes Log10 of the given expression.
- template<Expression Ex>
  constexpr auto log1p (Ex const &ex) noexcept

  Computes Log1p of the given expression.
- template<Expression Ex>
  constexpr auto log2 (Ex const &ex) noexcept

  Computes Log2 of the given expression.
- template<Expression Ex>
  constexpr auto lrint (Ex const &ex) noexcept

  Computes Lrint of the given expression.
- template<Expression Ex>
  constexpr auto lround (Ex const &ex) noexcept

  Computes Lround of the given expression.
- template<Expression Ex>
  constexpr auto nearbyint (Ex const &ex) noexcept

  Computes Nearbyint of the given expression.
- template<Expression Ex>
  constexpr auto rint (Ex const &ex) noexcept

  Computes Rint of the given expression.
- template<Expression Ex>
  constexpr auto round (Ex const &ex) noexcept

  Computes Round of the given expression.
- template<Expression Ex>
  constexpr auto sin (Ex const &ex) noexcept

  Computes Sin of the given expression.
- template<Expression Ex>
  constexpr auto sinh (Ex const &ex) noexcept

  Computes Sinh of the given expression.
- template<Expression Ex>
  constexpr auto sqrt (Ex const &ex) noexcept

  Computes Sqrt of the given expression.
- template<Expression Ex>
  constexpr auto tan (Ex const &ex) noexcept

  Computes Tan of the given expression.
- template<Expression Ex>
  constexpr auto tanh (Ex const &ex) noexcept

  Computes Tanh of the given expression.
- template<Expression Ex>
  constexpr auto trunc (Ex const &ex) noexcept

  Computes Trunc of the given expression.
- template<Expression Lhs_Expression, Variable Rhs_Expression>
  constexpr auto assign (Lhs_Expression const &lhs_ex, Rhs_Expression const &rhs_ex) noexcept
- template<Expression Ex>
  auto poisson (Ex const &ex) noexcept

Variables

- template<class T >
  constexpr bool ceras::is_unary_operator_v = is_unary_operator<T>::value
- template<class T >
  constexpr bool ceras::is_binary_operator_v = is_binary_operator<T>::value
- *auto y = variable<tensor<float>>{ }
- *auto sqr = hypot(x, y)

9.23.1 Function Documentation

9.23.1.1 abs()

```
template<Expression Ex>
constexpr auto abs (
            Ex const & ex )  [constexpr], [noexcept]
```

Computes Abs of the given expression.

Example code:
```
auto a = variable{ random<float>( {2, 3, 5} ) };
auto b = abs( a );
```

9.23.1.2 acos()

```
template<Expression Ex>
constexpr auto acos (
            Ex const & ex )  [constexpr], [noexcept]
```

Computes Acos of the given expression.

Example code:
```
auto a = variable{ random<float>( {2, 3, 5} ) };
auto b = acos( a );
```

9.23.1.3 acosh()

```
template<Expression Ex>
constexpr auto acosh (
            Ex const & ex )  [constexpr], [noexcept]
```

Computes Acosh of the given expression.

Example code:
```
auto a = variable{ random<float>( {2, 3, 5} ) };
auto b = acosh( a );
```

9.23.1.4 argmax()

```
auto argmax (
            unsigned long axis = 0 ) [inline], [noexcept]
```

Returns the index with the largest value across axes of an input tensor.

```
auto a = variable{ ... };
auto ma = argmax( 1 )( a );
```

9.23.1.5 argmin()

```
auto argmin (
            unsigned long axis = 0 ) [inline], [noexcept]
```

Returns the index with the smallest value across axes of an input tensor.

```
auto a = variable{ ... };
auto ma = argmin( 1 )( a );
```

9.23.1.6 asin()

```
template<Expression Ex>
constexpr auto asin (
            Ex const & ex ) [constexpr], [noexcept]
```

Computes Asin of the given expression.

Example code:

```
auto a = variable{ random<float>( {2, 3, 5} ) };
auto b = asin( a );
```

9.23.1.7 asinh()

```
template<Expression Ex>
constexpr auto asinh (
            Ex const & ex ) [constexpr], [noexcept]
```

Computes Asinh of the given expression.

Example code:

```
auto a = variable{ random<float>( {2, 3, 5} ) };
auto b = asinh( a );
```

9.23.1.8 assign()

```
template<Expression Lhs_Expression, Variable Rhs_Expression>
constexpr auto assign (
            Lhs_Expression const & lhs_ex,
            Rhs_Expression const & rhs_ex ) [constexpr], [noexcept]
```

@breif Updating the second expression's value by assining the first one to it. The second expression should be a 'variable'.

Parameters

lhs_ex	A mutable value.
rhs_ex	An expression to be assigned to lhs_ex. TODO: Fixme, this implementation is wrong

```
auto x = constant{ ... } * constant{ ... };
auto v = variable{ ... };
assgin( x, v );
```

9.23.1.9 atan()

```
template<Expression Ex>
constexpr auto atan (
            Ex const & ex ) [constexpr], [noexcept]
```

Computes Atan of the given expression.

Example code:
```
auto a = variable{ random<float>( {2, 3, 5} ) };
auto b = atan( a );
```

9.23.1.10 atan2()

```
template<Expression Lhs_Expression, Expression Rhs_Expression>
constexpr auto atan2 (
            Lhs_Expression const & lhs_ex,
            Rhs_Expression const & rhs_ex ) [constexpr], [noexcept]
```

Computes the arc tangent of y/x using the signs of arguments to determine the correct quadrant.

9.23.1.11 atanh()

```
template<Expression Ex>
constexpr auto atanh (
            Ex const & ex ) [constexpr], [noexcept]
```

Computes Atanh of the given expression.

Example code:
```
auto a = variable{ random<float>( {2, 3, 5} ) };
auto b = atanh( a );
```

9.23.1.12 average_pooling_2d()

```
auto average_pooling_2d (
            unsigned long stride ) [inline], [noexcept]
```

9.23.1.13 batch_normalization()

```
template<typename T >
requires std::floating_point<T>
auto batch_normalization (
            T const momentum = 0.98 )  [inline], [noexcept]
```

9.23.1.14 cbrt()

```
template<Expression Ex>
constexpr auto cbrt (
            Ex const & ex )  [constexpr], [noexcept]
```

Computes Cbert of the given expression.

Example code:
```
auto a = variable{ random<float>( {2, 3, 5} ) };
auto b = cbrt( a );
```

9.23.1.15 ceil()

```
template<Expression Ex>
constexpr auto ceil (
            Ex const & ex )  [constexpr], [noexcept]
```

Computes Ceil of the given expression.

Example code:
```
auto a = variable{ random<float>( {2, 3, 5} ) };
auto b = ceil( a );
```

9.23.1.16 clip()

```
template<typename Float >
requires std::floating_point<Float>
constexpr auto clip (
            Float lower,
            Float upper = std::numeric_limits<Float>::max() )  [constexpr], [noexcept]
```

9.23.1.17 concat() [1/2]

```
template<Expression Lhs_Expression, Expression Rhs_Expression>
constexpr auto concat (
            Lhs_Expression const & lhs_ex,
            Rhs_Expression const & rhs_ex )  [constexpr], [noexcept]
```

9.23.1.18 concat() [2/2]

```
auto concat (
            unsigned long axe = -1 )  [inline]
```

9.23.1.19 concatenate() [1/2]

```
template<Expression Lhs_Expression, Expression Rhs_Expression>
constexpr auto concatenate (
            Lhs_Expression const & lhs_ex,
            Rhs_Expression const & rhs_ex )  [constexpr], [noexcept]
```

9.23.1.20 concatenate() [2/2]

```
auto concatenate (
            unsigned long axe = -1 )  [inline]
```

9.23.1.21 conv2d()

```
auto conv2d (
            unsigned long row_input,
            unsigned long col_input,
            unsigned long const row_stride = 1,
            unsigned long const col_stride = 1,
            unsigned long const row_dilation = 1,
            unsigned long const col_dilation = 1,
            std::string const & padding = "valid" )  [inline], [noexcept]
```

9.23.1.22 cos()

```
template<Expression Ex>
constexpr auto cos (
            Ex const & ex )  [constexpr], [noexcept]
```

Computes Cos of the given expression.

Example code:
```
auto a = variable{ random<float>( {2, 3, 5} ) };
auto b = cos( a );
```

9.23.1.23 cosh()

```
template<Expression Ex>
constexpr auto cosh (
            Ex const & ex )  [constexpr], [noexcept]
```

Computes Cosh of the given expression.

Example code:
```
auto a = variable{ random<float>( {2, 3, 5} ) };
auto b = cosh( a );
```

9.23.1.24 cropping_2d()

```
auto cropping_2d (
            std::vector< unsigned long > const & padding )  [inline], [noexcept]
```

Cropping layer for 2D input. The input should have 4-dimensions: `(batch_size, row, col, channel)`. The output has 4-dimensions: `(batch_size, new_row, new_col, channel)`.

Parameters

padding	If a single integer, then apply symmetric cropping to height and width. If two integers, then first is for height and the second is for width. If four integers, then is intepreted as<tt>(top_crop, bottom_crop, left_crop, right_crop).

Example code:
```
auto a = variable{ random<float>( {32, 32, 3} ) };
auto b = cropping_2d( {8,} )( a ); // shape for b is (32-8-8, 32-8-8, 3)
auto c = cropping_2d( {8, 4} )( a ); // shape for c is (32-8-8, 32-4-4, 3)
auto d = cropping_2d( {8, 4, 2, 1} )( a ); // shape for d is (32-8-4, 32-2-1, 3)
```

9.23.1.25 drop_out()

```
template<typename T >
requires std::floating_point<T>
auto drop_out (
            T const factor )  [inline], [noexcept]
```

9.23.1.26 dropout()

```
template<typename T >
requires std::floating_point<T>
auto dropout (
            T const factor )  [inline], [noexcept]
```

dropout is an alias name of drop_out.

9.23.1.27 equal()

```
template<Expression Lhs_Expression, Expression Rhs_Expression, std::floating_point FP>
constexpr auto equal (
            Lhs_Expression const & lhs_ex,
            Rhs_Expression const & rhs_ex,
            FP threshold = 0.5 )  [constexpr], [noexcept]
```

Returns the truth value of (lhs == rhs) element-wise. [+1 for true, 0 for false]

Parameters

lhs_ex	The first operator.
rhs_ex	The second operator.

Returns

An instance of a binary operator that evaluate the element-wise equality of two input operators.

Example code:

```
auto l = variable<tensor<float>>{ /*...*/ };
auto r = place_holder<tensor<float>>{};
auto eq = equal(l, r);
```

9.23.1.28 erf()

```
template<Expression Ex>
constexpr auto erf (
            Ex const & ex )  [constexpr], [noexcept]
```

Computes Erf of the given expression.

Example code:

```
auto a = variable{ random<float>( {2, 3, 5} ) };
auto b = erf( a );
```

9.23.1.29 erfc()

```
template<Expression Ex>
constexpr auto erfc (
            Ex const & ex )  [constexpr], [noexcept]
```

Computes Erfc of the given expression.

Example code:

```
auto a = variable{ random<float>( {2, 3, 5} ) };
auto b = erfc( a );
```

9.23.1.30 exp()

```
template<Expression Ex>
constexpr auto exp (
        Ex const & ex )  [constexpr], [noexcept]
```

Computes Exp of the given expression.

Example code:
```
auto a = variable{ random<float>( {2, 3, 5} ) };
auto b = exp( a );
```

9.23.1.31 exp2()

```
template<Expression Ex>
constexpr auto exp2 (
        Ex const & ex )  [constexpr], [noexcept]
```

Computes Exp2 of the given expression.

Example code:
```
auto a = variable{ random<float>( {2, 3, 5} ) };
auto b = exp2( a );
```

9.23.1.32 expand_dims()

```
constexpr auto expand_dims (
        int axis = -1 )  [inline], [constexpr], [noexcept]
```

Expand input tensor with a length 1 axis inserted at index axis.
```
auto x = variable<float>{ {2, 3, 4}}
auto x0 = expand_dims(0)( x ); // new shape is ( 1, 2, 3, 4 )
auto x1 = expand_dims(1)( x ); // new shape is ( 2, 1, 3, 4 )
auto x2 = expand_dims(2)( x ); // new shape is ( 2, 3, 1, 4 )
auto x3 = expand_dims(-1)( x ); // new shape is ( 2, 3, 4, 1 )
```

9.23.1.33 expm1()

```
template<Expression Ex>
constexpr auto expm1 (
        Ex const & ex )  [constexpr], [noexcept]
```

Computes Expm1 of the given expression.

Example code:
```
auto a = variable{ random<float>( {2, 3, 5} ) };
auto b = expm1( a );
```

9.23.1.34 fabs()

```
template<Expression Ex>
constexpr auto fabs (
            Ex const & ex )  [constexpr], [noexcept]
```

Computes Fabs of the given expression.

Example code:
```
auto a = variable{ random<float>( {2, 3, 5} ) };
auto b = fabs( a );
```

9.23.1.35 flatten()

```
template<Expression Ex>
constexpr auto flatten (
            Ex const & ex )  [constexpr], [noexcept]
```

Flatten input tensor.
```
auto x = .....; // an operator returns tensor of shape ( 12, 34, 1 2 )
auto f = flatten( x ); // returns tensor of shape (12*34*1*2, )
```

9.23.1.36 floor()

```
template<Expression Ex>
constexpr auto floor (
            Ex const & ex )  [constexpr], [noexcept]
```

Computes Floor of the given expression.

Example code:
```
auto a = variable{ random<float>( {2, 3, 5} ) };
auto b = floor( a );
```

9.23.1.37 general_conv2d()

```
auto general_conv2d (
            unsigned long const row_stride = 1,
            unsigned long const col_stride = 1,
            unsigned long const row_dilation = 1,
            unsigned long const col_dilation = 1,
            std::string const & padding = "valid" )  [inline], [noexcept]
```

Conv2D not constrained by the input shape.

9.23.1.38 hypot()

```
template<Expression Ex, Expression Ey>
*endcode **constexpr auto hypot (
            Ex const & ex,
            Ey const & ey )  [constexpr], [noexcept]
```

9.23.1.39 identity()

```
template<Expression Ex>
constexpr auto identity (
            Ex const & ex )  [constexpr], [noexcept]
```

Identity operation.

9.23.1.40 img2col()

```
auto img2col (
            unsigned long const row_kernel,
            unsigned long col_kernel = -1,
            unsigned long const row_padding = 0,
            unsigned long col_padding = 0,
            unsigned long const row_stride = 1,
            unsigned long const col_stride = 1,
            unsigned long const row_dilation = 1,
            unsigned long const col_dilation = 1 )  [inline], [noexcept]
```

9.23.1.41 llrint()

```
template<Expression Ex>
constexpr auto llrint (
            Ex const & ex )  [constexpr], [noexcept]
```

Computes Llrint of the given expression.

Example code:
```
auto a = variable{ random<float>( {2, 3, 5} ) };
auto b = llrint( a );
```

9.23.1.42 llround()

```
template<Expression Ex>
constexpr auto llround (
            Ex const & ex ) [constexpr], [noexcept]
```

Computes Llround of the given expression.

Example code:
```
auto a = variable{ random<float>( {2, 3, 5} ) };
auto b = llround( a );
```

9.23.1.43 log()

```
template<Expression Ex>
constexpr auto log (
            Ex const & ex ) [constexpr], [noexcept]
```

Computes Log of the given expression.

Example code:
```
auto a = variable{ random<float>( {2, 3, 5} ) };
auto b = log( a );
```

9.23.1.44 log10()

```
template<Expression Ex>
constexpr auto log10 (
            Ex const & ex ) [constexpr], [noexcept]
```

Computes Log10 of the given expression.

Example code:
```
auto a = variable{ random<float>( {2, 3, 5} ) };
auto b = log10( a );
```

9.23.1.45 log1p()

```
template<Expression Ex>
constexpr auto log1p (
            Ex const & ex ) [constexpr], [noexcept]
```

Computes Log1p of the given expression.

Example code:
```
auto a = variable{ random<float>( {2, 3, 5} ) };
auto b = log1p( a );
```

9.23.1.46 log2()

```
template<Expression Ex>
constexpr auto log2 (
            Ex const & ex )  [constexpr], [noexcept]
```

Computes Log2 of the given expression.

Example code:
```
auto a = variable{ random<float>( {2, 3, 5} ) };
auto b = log2( a );
```

9.23.1.47 lrint()

```
template<Expression Ex>
constexpr auto lrint (
            Ex const & ex )  [constexpr], [noexcept]
```

Computes Lrint of the given expression.

Example code:
```
auto a = variable{ random<float>( {2, 3, 5} ) };
auto b = lrint( a );
```

9.23.1.48 lround()

```
template<Expression Ex>
constexpr auto lround (
            Ex const & ex )  [constexpr], [noexcept]
```

Computes Lround of the given expression.

Example code:
```
auto a = variable{ random<float>( {2, 3, 5} ) };
auto b = lround( a );
```

9.23.1.49 max_pooling_2d()

```
auto max_pooling_2d (
            unsigned long stride )  [inline], [noexcept]
```

9.23.1.50 maximum()

```
template<Expression Lhs_Expression, Expression Rhs_Expression>
constexpr auto maximum (
            Lhs_Expression const & lhs_ex,
            Rhs_Expression const & rhs_ex )  [constexpr], [noexcept]
```

9.23.1.51 minimum()

```
template<Expression Lhs_Expression, Expression Rhs_Expression>
constexpr auto minimum (
            Lhs_Expression const & lhs_ex,
            Rhs_Expression const & rhs_ex )  [constexpr], [noexcept]
```

9.23.1.52 nearbyint()

```
template<Expression Ex>
constexpr auto nearbyint (
            Ex const & ex )  [constexpr], [noexcept]
```

Computes Nearbyint of the given expression.

Example code:

```
auto a = variable{ random<float>( {2, 3, 5} ) };
auto b = nearbyint( a );
```

9.23.1.53 normalization_batch()

```
template<typename T = double>
requires std::floating_point<T>
auto normalization_batch (
            T const momentum = 0.98 )  [inline], [noexcept]
```

9.23.1.54 ones_like()

```
template<Expression Ex>
auto ones_like (
            Ex const & ex )  [noexcept]
```

`ones_like` produces a tensor of the same shape as the input expression, but with every element to be 1.

Returns

An unary operator that takes an unary operator, and producing an output tensor Example Code:

```
auto va = variable{ ones<float>({3, 3, 3}) };
auto v_rand = ones_like( va ); // this expression will produces a tensor of shape (3, 3, 3), with every
element to be 1.
```

9.23.1.55 poisson()

```
template<Expression Ex>
auto poisson (
            Ex const & ex ) [noexcept]
```

`poisson` produces random tensor from a normal distribution

Returns

An unary operator that takes an unary operator, and producing output tensor subjects to a Poisson distribution. The shape of the output tensor has the same shape corresponding to the input unary operator.

Example Code
```
auto va = variable{ ones<float>({3, 3, 3}) };
auto v_rand = poisson( va ); // this expression will produces a tensor of shape (3, 3, 3) subjects to a
      Poisson distribution
```

9.23.1.56 random_normal_like()

```
template<typename T = float>
requires std::floating_point<T>
auto random_normal_like (
            T mean = 0.0,
            T stddev = 1.0 ) [inline], [noexcept]
```

`random_normal_like` produces random tensor from a normal distribution

Parameters

mean	Mean of the normal distribution, a scalar.
stddev	Standard deviation of the normal distribution, a scalar.

Returns

An unary operator that takes an unary operator, and producing output tensor from a normal distribution. The shape of the output tensor has the same shape corresponding to the input unary operator.

Example Code
```
auto va = variable{ ones<float>({3, 3, 3}) };
auto v_rand = random_normal_like( 1.0, 4.0 )( va ); // this expression will produces a tensor of shape (3,
      3, 3) from a normal distribution with parameters (1.0, 4.0)
```

9.23.1.57 reduce_max()

```
auto reduce_max (
            unsigned long axis = -1 ) [inline], [noexcept]
```

Reduce maximum elements along an axis.

Parameters

axis	The axis along which to reduce maximum values. Defaults to the last axis.

Example code:
```
auto a = variable{ random<float>( {2, 3, 5} ) };
auto b = reduce_max( 0 )( a ); // <- output shape is ( 3, 5 )
auto b = reduce_max( 1 )( a ); // <- output shape is ( 2, 5 )
auto b = reduce_max( 2 )( a ); // <- output shape is ( 2, 3 )
auto b = reduce_max( )( a ); // <- output shape is ( 2, 3 )
```

9.23.1.58 reduce_min()

```
auto reduce_min (
            unsigned long axis = -1 ) [inline], [noexcept]
```

Reduce minimal elements along an axis.

Parameters

axis	The axis along which to reduce minimal values. Defaults to the last axis.

Example code:
```
auto a = variable{ random<float>( {2, 3, 5} ) };
auto b = reduce_min( 0 )( a ); // <- output shape is ( 3, 5 )
auto b = reduce_min( 1 )( a ); // <- output shape is ( 2, 5 )
auto b = reduce_min( 2 )( a ); // <- output shape is ( 2, 3 )
auto b = reduce_min( )( a ); // <- output shape is ( 2, 3 )
```

9.23.1.59 reduce_sum()

```
auto reduce_sum (
            unsigned long axis ) [inline], [noexcept]
```

Reduce sum elements along an axis.

Parameters

axis	The axis along which to reduce sum.

Example code:
```
auto a = variable{ random<float>( {2, 3, 5} ) };
auto b = reduce_sum( 0 )( a ); // <- output shape is ( 3, 5 )
auto b = reduce_sum( 1 )( a ); // <- output shape is ( 2, 5 )
auto b = reduce_sum( 2 )( a ); // <- output shape is ( 2, 3 )
auto b = reduce_sum( -1 )( a ); // <- output shape is ( 2, 3 )
```

9.23.1.60 repeat()

```
auto repeat (
            unsigned long repeats,
            unsigned long axis = -1 )  [inline], [noexcept]
```

Repeats elements along an axis.

Parameters

repeats	The number of repetitions for each element.
axis	The axis along which to repeat values. Defaults to the last axis.

Example code:
```
auto a = variable{ random<float>( {2, 3, 5} };
auto b0 = repeat( 2, 0 )( a ); // <- output shape is ( 4, 3, 5 )
auto b1 = repeat( 2, 1 )( a ); // <- output shape is ( 2, 6, 5 )
auto b2 = repeat( 2, 2 )( a ); // <- output shape is ( 2, 3, 10 )
auto bx = repeat( 2 )( a ); // <- output shape is ( 2, 3, 10 )
```

9.23.1.61 reshape()

```
auto reshape (
            std::vector< unsigned long > const & new_shape,
            bool include_batch_flag = true )  [inline], [noexcept]
```

9.23.1.62 rint()

```
template<Expression Ex>
constexpr auto rint (
            Ex const & ex )  [constexpr], [noexcept]
```

Computes Rint of the given expression.

Example code:
```
auto a = variable{ random<float>( {2, 3, 5} ) };
auto b = rint( a );
```

9.23.1.63 round()

```
template<Expression Ex>
constexpr auto round (
            Ex const & ex )  [constexpr], [noexcept]
```

Computes Round of the given expression.

Example code:
```
auto a = variable{ random<float>( {2, 3, 5} ) };
auto b = round( a );
```

9.23.1.64 sign()

```
template<Expression Ex>
constexpr auto sign (
            Ex const & ex ) [constexpr], [noexcept]
```

Returns the sign. [1 for positive, 0 for 0 and -1 for negative]

Parameters

ex	The input operator.

Returns

An instance of a unary_operator that evaluate the sign of the input operator.

Example code:

```
auto e = variable<tensor<float>>{ /*...*/ };
auto si = sign(e);
```

9.23.1.65 sin()

```
template<Expression Ex>
constexpr auto sin (
            Ex const & ex ) [constexpr], [noexcept]
```

Computes Sin of the given expression.

Example code:

```
auto a = variable{ random<float>( {2, 3, 5} ) };
auto b = sin( a );
```

9.23.1.66 sinh()

```
template<Expression Ex>
constexpr auto sinh (
            Ex const & ex ) [constexpr], [noexcept]
```

Computes Sinh of the given expression.

Example code:

```
auto a = variable{ random<float>( {2, 3, 5} ) };
auto b = sinh( a );
```

9.23.1.67 sliding_2d()

```
auto sliding_2d (
            unsigned long pixels )  [inline], [noexcept]
```

9.23.1.68 sqrt()

```
template<Expression Ex>
constexpr auto sqrt (
            Ex const & ex )  [constexpr], [noexcept]
```

Computes Sqrt of the given expression.

Example code:
```
auto a = variable{ random<float>( {2, 3, 5} ) };
auto b = sqrt( a );
```

9.23.1.69 tan()

```
template<Expression Ex>
constexpr auto tan (
            Ex const & ex )  [constexpr], [noexcept]
```

Computes Tan of the given expression.

Example code:
```
auto a = variable{ random<float>( {2, 3, 5} ) };
auto b = tan( a );
```

9.23.1.70 tanh()

```
template<Expression Ex>
constexpr auto tanh (
            Ex const & ex )  [constexpr], [noexcept]
```

Computes Tanh of the given expression.

Example code:
```
auto a = variable{ random<float>( {2, 3, 5} ) };
auto b = tanh( a );
```

9.23.1.71 transpose()

```
template<Expression Ex>
auto transpose (
            Ex const & ex ) [noexcept]
```

Transpose a matrix.

9.23.1.72 trunc()

```
template<Expression Ex>
constexpr auto trunc (
            Ex const & ex ) [constexpr], [noexcept]
```

Computes Trunc of the given expression.

Example code:
```
auto a = variable{ random<float>( {2, 3, 5} ) };
auto b = trunc( a );
```

9.23.1.73 up_sampling_2d()

```
auto up_sampling_2d (
            unsigned long stride ) [inline], [noexcept]
```

9.23.1.74 upsampling_2d()

```
auto upsampling_2d (
            unsigned long stride ) [inline], [noexcept]
```

9.23.1.75 zero_padding_2d()

```
auto zero_padding_2d (
            std::vector< unsigned long > const & padding ) [inline], [noexcept]
```

Zero-padding layer for 2D input. The input should have 4-dimensions: `(batch_size, row, col, channel)`. The output has 4-dimensions: `(batch_size, new_row, new_col, channel)`.

Parameters

padding	If a single integer, then apply symmetric padding to height and width. If two integers, then first is for height and the second is for width. If four integers, then is intepreted as<tt>(top_pad, bottom_pad, left_pad, right_pad).

Example code:
```
auto a = variable{ random<float>( {16, 16, 3} ) };
auto b = zero_padding_2d( {8,} )( a ); // shape for b is (8+16+8, 8+16+8, 3)
auto c = zero_padding_2d( {8, 4} )( a ); // shape for c is (8+16+8, 4+16+4, 3)
auto d = zero_padding_2d( {8, 4, 2, 1} )( a ); // shape for d is (8+16+4, 2+16+1, 3)
```

9.23.1.76 zeros_like()

```
template<Expression Ex>
auto zeros_like (
            Ex const & ex )  [noexcept]
```

`zeros_like` produces a tensor of the same shape as the input expression, but with every element to be 0.

Returns

An unary operator that takes an unary operator, and producing an output tensor Example Code:
```
auto va = variable{ ones<float>({3, 3, 3}) };
auto v_rand = zeros_like( va ); // this expression will produces a tensor of shape (3, 3, 3), with
every element to be 0.
```

9.23.2 Variable Documentation

9.23.2.1 sqr

```
* auto sqr = hypot( x, y )
```

9.23.2.2 y

```
* auto y = variable<tensor<float>>{ }
```

9.24 operation.hpp

Go to the documentation of this file.
```
#ifndef IPKVWSJOCMGGVRASCBLPYHFBCHRIVEXYBOMMDAKFAUDFYVYOOOISLRXJNUJKPJEVMLDPRDSNM
#define IPKVWSJOCMGGVRASCBLPYHFBCHRIVEXYBOMMDAKFAUDFYVYOOOISLRXJNUJKPJEVMLDPRDSNM

#include "./includes.hpp"
#include "./place_holder.hpp"
#include "./variable.hpp"
#include "./constant.hpp"
#include "./value.hpp"
#include "./utils/range.hpp"
#include "./utils/debug.hpp"
#include "./config.hpp"
#include "./utils/context_cast.hpp"
#include "./utils/for_each.hpp"
#include "./utils/id.hpp"
#include "./utils/enable_shared.hpp"
#include "./utils/fmt.hpp"

```

```
namespace ceras
{

    struct identity_output_shape_calculator
    {
        std::vector<unsigned long> operator()( std::vector<unsigned long> const& input_shape ) const noexcept
        {
            return input_shape;
        }

        std::vector<unsigned long> operator()( std::vector<unsigned long> const& lhs_input_shape, std::vector<unsigned long> const& rhs_input_shape ) const noexcept
        {
            return lhs_input_shape.size() > rhs_input_shape.size() ?  lhs_input_shape :  rhs_input_shape;
        }

        std::vector<unsigned long> operator()() const noexcept
        {
            return std::vector<unsigned long>{ {-1UL,} };
        }
    }; // struct identity_output_shape_calculator


    template< typename Operator, typename Forward_Action, typename Backward_Action, typename Output_Shape_Calculator = identity_output_shape_calculator >
    struct unary_operator :  enable_id<unary_operator<Operator, Forward_Action, Backward_Action>, "Unary Operator">
    {
        Operator op_;
        Forward_Action forward_action_;
        Backward_Action backward_action_;
        Output_Shape_Calculator output_shape_calculator_;

        typedef decltype( std::declval<Forward_Action>()( std::declval<decltype(op_)>().forward() ) ) tensor_type;

        tensor_type input_data_;
        tensor_type output_data_;

        unary_operator( Operator const& op, Forward_Action const& forward_action, Backward_Action const& backward_action, Output_Shape_Calculator const& output_shape_calculator ) noexcept :
            op_{op}, forward_action_{ forward_action }, backward_action_{ backward_action }, output_shape_calculator_{ output_shape_calculator } { }

        auto forward()
        {
            input_data_ = op_.forward();
            output_data_ = forward_action_( input_data_ );
            return output_data_;
        }

        void backward( tensor_type const& grad )
        {
            auto const& current_gradient = backward_action_( input_data_, output_data_, grad );
            op_.backward( current_gradient );
        }

        std::vector<unsigned long> shape() const noexcept
        {
            return output_shape_calculator_( op_.shape() );
        }

    };

    template< typename Forward_Action, typename Backward_Action, typename Output_Shape_Calculator= identity_output_shape_calculator >
    auto constexpr make_unary_operator( Forward_Action const& unary_forward_action,
                                        Backward_Action const& unary_backward_action,
                                        std::string const& name = "Anonymous Unary Operator",
                                        Output_Shape_Calculator const& output_shape_calculator = Output_Shape_Calculator{} ) noexcept
    {
        return [&]( auto const& op ) noexcept
        {
            auto ans = unary_operator{ op, unary_forward_action, unary_backward_action, output_shape_calculator };
            ans.name_ = name;
            return ans;
        };
    }


    template< typename Lhs_Operator, typename Rhs_Operator, typename Forward_Action, typename Backward_Action, typename Output_Shape_Calculator= identity_output_shape_calculator >
    struct binary_operator :enable_id<binary_operator<Lhs_Operator, Rhs_Operator, Forward_Action, Backward_Action>, "Binary Operator">
```

```
{
    Lhs_Operator lhs_op_;
    Rhs_Operator rhs_op_;
    Forward_Action forward_action_;
    Backward_Action backward_action_; // backward action for binary operator produces a tuple of two tensors
    Output_Shape_Calculator output_shape_calculator_;

    typedef typename tensor_deduction<Lhs_Operator, Rhs_Operator>::tensor_type tensor_type; // defined in value.hpp

    tensor_type lhs_input_data_;
    tensor_type rhs_input_data_;
    tensor_type output_data_;

    binary_operator( Lhs_Operator const& lhs_op, Rhs_Operator const& rhs_op, Forward_Action const& forward_action, Backward_Action const& backward_action, Output_Shape_Calculator const& output_shape_calculator ) noexcept :
        lhs_op_{lhs_op}, rhs_op_{rhs_op}, forward_action_{ forward_action }, backward_action_{ backward_action }, output_shape_calculator_{ output_shape_calculator } { }

    auto forward()
    {
        static_assert( !(is_value_v<Lhs_Operator> && is_value_v<Rhs_Operator>), "Not valid for two values" );

        if constexpr ( is_value_v<Lhs_Operator> )
        {
            rhs_input_data_ = rhs_op_.forward();
            lhs_input_data_ = lhs_op_.forward( rhs_input_data_ );
        }
        else if constexpr ( is_value_v<Rhs_Operator> )
        {
            lhs_input_data_ = lhs_op_.forward();
            rhs_input_data_ = rhs_op_.forward( lhs_input_data_ );
        }
        else
        {
            lhs_input_data_ = lhs_op_.forward();
            rhs_input_data_ = rhs_op_.forward();
        }
        output_data_ = forward_action_( lhs_input_data_, rhs_input_data_ );
        return output_data_;
    }

    void backward( tensor_type const& grad )
    {
        auto const& [current_gradient_lhs, current_gradient_rhs] = backward_action_( lhs_input_data_, rhs_input_data_, output_data_, grad );
        lhs_op_.backward( current_gradient_lhs );
        rhs_op_.backward( current_gradient_rhs );
    }

    std::vector<unsigned long> shape() const noexcept
    {
        if constexpr ( is_value_v<Lhs_Operator> )
            return rhs_op_.shape();
        else if constexpr ( is_value_v<Rhs_Operator> )
            return lhs_op_.shape();
        else
            return output_shape_calculator_( lhs_op_.shape(), rhs_op_.shape() );
    }

};

template< typename Forward_Action, typename Backward_Action, typename Output_Shape_Calculator= identity_output_shape_calculator >
auto make_binary_operator( Forward_Action const& binary_forward_action,
                           Backward_Action const& binary_backward_action,
                           std::string const& name = "Anonymous Binary Operator",
                           Output_Shape_Calculator const& output_shape_calculator = Output_Shape_Calculator{} ) noexcept
{
    return [&]( auto const& lhs_op, auto const& rhs_op ) noexcept
    {
        auto ans = binary_operator{ lhs_op, rhs_op, binary_forward_action, binary_backward_action, output_shape_calculator };
        ans.name_ = name;
        return ans;
    };
}


template< typename T >
struct is_unary_operator :  std::false_type{};

template< typename Operator, typename Forward_Action, typename Backward_Action, typename
```

```
Output_Shape_Calculator >
  struct is_unary_operator< unary_operator<Operator, Forward_Action, Backward_Action, Output_Shape_Calculator> > :  std::true_type {};

  template< class T >
  inline constexpr bool is_unary_operator_v = is_unary_operator<T>::value;

  template< typename T >
  concept Unary_Operator = is_unary_operator_v<T>;


  template< typename T >
  struct is_binary_operator :  std::false_type{};

  template< typename Lhs_Operator, typename Rhs_Operator, typename Forward_Action, typename Backward_Action, typename Output_Shape_Calculator >
  struct is_binary_operator< binary_operator<Lhs_Operator, Rhs_Operator, Forward_Action, Backward_Action, Output_Shape_Calculator> > :  std::true_type {};

  template< class T >
  inline constexpr bool is_binary_operator_v = is_binary_operator<T>::value;

  template< typename T >
  concept Binary_Operator = is_binary_operator_v<T>;

  template< typename T >
  concept Operator = Unary_Operator<T> || Binary_Operator<T>;

  template< typename T >
  concept Expression = Operator<T> || Variable<T> || Place_Holder<T> || Constant<T> || Value<T>;


  template< Expression Ex >
  inline std::string computation_graph( Ex const& ex ) noexcept
  {
      auto generate_node_and_label = []<Expression Expr>( Expr const& expr ) noexcept
      {
          std::string const id = std::to_string( expr.id() );
          std::string const name = expr.name();
          std::string node = std::string{"n"} + id;

          std::vector<long long> shape;
          {
              std::vector<unsigned long> _shape = expr.shape();
              shape.resize( _shape.size() );
              std::copy( _shape.begin(), _shape.end(), shape.begin() );
              if ( _shape.size() > 0 && _shape[0] == -1UL )
                  shape[0] = -1;
          }

          std::string label = fmt::format( "{} <shape:{}> [id:{}]", name, shape, id);
          return std::make_tuple( node, label );
      };

      auto generate_dot = [&generate_node_and_label]<Expression Expr>( Expr const& expr, auto const& _generate_dot ) noexcept
      {
          auto const& [node, label] = generate_node_and_label( expr );
          std::string const& expr_dot = node + std::string{" [label=\""} + label + std::string{"\"] ;\n"};

          if constexpr( is_unary_operator_v<Expr> )
          {
              auto const& [n_node, n_label] = generate_node_and_label( expr.op_ );
              std::string const& arrow_relation = n_node + std::string{" -> "} + node + std::string{" ;\n"};
              std::string const& op_dot = _generate_dot( expr.op_, _generate_dot );
              return expr_dot + arrow_relation + op_dot;
          }
          else if constexpr( is_binary_operator_v<Expr> )
          {
              // for LHS operator
              auto const& [n_lhs_node, n_lhs_label] = generate_node_and_label( expr.lhs_op_ );
              std::string const& arrow_lhs_relation = n_lhs_node + std::string{" -> "} + node + std::string{" ;\n"};
              std::string const& op_lhs_dot = _generate_dot( expr.lhs_op_, _generate_dot );

              // for RHS operator
              auto const& [n_rhs_node, n_rhs_label] = generate_node_and_label( expr.rhs_op_ );
              std::string const& arrow_rhs_relation = n_rhs_node + std::string{" -> "} + node + std::string{" ;\n"};
              std::string const& op_rhs_dot = _generate_dot( expr.rhs_op_, _generate_dot );

              return expr_dot + arrow_lhs_relation + arrow_rhs_relation + op_lhs_dot + op_rhs_dot;
          }
          else if constexpr ( is_variable_v<Expr> )
          {
```

```
                std::vector<unsigned long> const& shape = expr.shape();
                bool const training_state = expr.trainable();

                // shape
                std::stringstream ss;
                std::copy( shape.begin(), shape.end(), std::ostream_iterator<unsigned long>( ss, " " ) );
                std::string const& str_shape = ss.str() + (training_state ?  std::string{"), trainable"} :  std::string{"), non-trainable"});
                // trainable state
                std::string const& new_label = label + std::string{"[("} + str_shape + std::string{"]"};

                if (!training_state)
                    return node + std::string{" [shape=box,label=\""} + new_label + std::string{"\"] ;\n"};

                return node + std::string{" [peripheries=3,style=filled,color=\".7 .3 1.0\",shape=box,label=\""} + new_label + std::string{"\"] ;\n"};
            }
            else
            {
                return expr_dot;
            }
        };

        std::string const& head = "\n\ndigraph g {\n";
        std::string const& tail = "}\n\n";
        return head + generate_dot( ex, generate_dot ) + tail;
    }


    namespace
    {
        struct plus_context
        {
            auto make_forward() const noexcept
            {
                return []<Tensor Tsor>( Tsor const& lhs_tensor, Tsor const& rhs_tensor ) noexcept
                {
                    better_assert( !has_nan( lhs_tensor ), "forward propagation for operator plus: lhs_tensor contains Nan!" );
                    better_assert( !has_nan( rhs_tensor ), "forward propagation for operator plus: rhs_tensor contains Nan!" );
                    return add( lhs_tensor, rhs_tensor );
                };
            }

            auto const make_backward() const noexcept
            {
                return []<Tensor Tsor>( Tsor const& lhs_input, Tsor const& rhs_input, Tsor const&, Tsor const& grad ) noexcept
                {
                   better_assert( !has_nan( grad ), "backprop:  upcoming gradient for operator + contains NaN!" );

                   auto const& grad_fun = [&grad]( auto const& input )
                   {
                       Tsor ans = grad.deep_copy();
                       while( input.ndim() < ans.ndim() )
                           ans = sum( ans, 0 );
                       auto const& shape = input.shape();
                       for ( auto axis :  range( input.ndim() ) )
                           if ( shape[axis] == 1 )
                              ans = sum( ans, axis, true );
                       return ans;
                   };
                   return std::make_tuple( grad_fun( lhs_input), grad_fun( rhs_input ) );
                };
            }
        }; // plus_context
    }//anonymous namespace

    template< Expression Lhs_Expression, Expression Rhs_Expression >
    auto constexpr plus( Lhs_Expression const& lhs_ex, Rhs_Expression const& rhs_ex ) noexcept
    {
        auto const& shape_calculator = []( std::vector<unsigned long> const& l, std::vector<unsigned long> const& r ) noexcept
        {
            return broadcast_shape( l, r );
        };


        return make_binary_operator( plus_context{}.make_forward(), plus_context{}.make_backward(), "Plus", shape_calculator )( lhs_ex, rhs_ex );
    }

    template< Expression Lhs_Expression, Expression Rhs_Expression >
```

```
    auto constexpr operator + ( Lhs_Expression const& lhs_ex, Rhs_Expression const& rhs_ex ) noexcept
    {
        return plus( lhs_ex, rhs_ex );
    }

    template< Expression Ex >
    auto constexpr operator + ( Ex const& ex ) noexcept
    {
        return ex;
    }

    namespace
    {
        struct multiplication_context
        {
            auto make_forward() const noexcept
            {
                return []( std::shared_ptr<std::any> forward_cache ) noexcept
                {
                    return [forward_cache]<Tensor Tsor>( Tsor const& lhs_tensor, Tsor const& rhs_tensor ) noexcept
                    {
                        Tsor& ans = context_cast<Tsor>( forward_cache );
                        multiply( lhs_tensor, rhs_tensor, ans );
                        return ans;
                    };
                };
            }
            auto make_backward() const noexcept
            {
                return []( std::shared_ptr<std::any> backward_cache_lhs, std::shared_ptr<std::any> backward_cache_rhs ) noexcept
                {
                    return [backward_cache_lhs, backward_cache_rhs]<Tensor Tsor>( Tsor const& lhs_input, Tsor const& rhs_input, Tsor const&, Tsor const& grad ) noexcept
                    {
                       // left branch <-- grad * rhs^T
                       auto const& g_shape = grad.shape();
                       auto const[m, n] = std::make_tuple( g_shape[0], g_shape[1] ); // 4, 1
                       auto const k = *(lhs_input.shape().rbegin()); // 13

                       Tsor& lhs_grad = context_cast<Tsor>( backward_cache_lhs );
                       lhs_grad.resize( lhs_input.shape() );

                       gemm( grad.data(), false, rhs_input.data(), true, m, n, k, lhs_grad.data() );

                       // right branch <-- lhs^T * grad
                       Tsor& rhs_grad = context_cast<Tsor>( backward_cache_rhs );
                       rhs_grad.resize( rhs_input.shape() );
                       gemm( lhs_input.data(), true, grad.data(), false, k, m, n, rhs_grad.data() );

                       return std::make_tuple( lhs_grad, rhs_grad );
                    };
                };
            }
        };//multiplication_context
    }//anonymous namespace

    template< Expression Lhs_Expression, Expression Rhs_Expression >
    auto operator * ( Lhs_Expression const& lhs_ex, Rhs_Expression const& rhs_ex ) noexcept
    {


        // case of Value * Operator and Operator * Value
        if constexpr( is_value_v<Lhs_Expression> || is_value_v<Rhs_Expression> )
        {
            return elementwise_product( lhs_ex, rhs_ex );
        }
        else
        {
            auto const& shape_calculator = []( std::vector<unsigned long> const& l, std::vector<unsigned long> const& r ) noexcept
            {
                better_assert( l.size() == 2, fmt::format( "expecting l size of 2, but got {}", l.size() ) );
                better_assert( r.size() == 2, fmt::format( "expecting r size of 2, but got {}", r.size() ) );
                better_assert( l[1] == r[0], fmt::format( "expecting l[1] == r[0], but l[1]={}, r[0]={}", l[1], r[0] ) ); // TODO: what if unknown dimension???
                return std::vector<unsigned long>{ {l[0], r[1]} };
            };
            std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
            std::shared_ptr<std::any> backward_cache_lhs = std::make_shared<std::any>();
            std::shared_ptr<std::any> backward_cache_rhs = std::make_shared<std::any>();
            return make_binary_operator( multiplication_context{}.make_forward()(forward_cache), multiplication_context{}.make_backward()(backward_cache_lhs, backward_cache_rhs), "Multiply", shape_calculator )( lhs_ex, rhs_ex );
```

```
        }
    }


    template <Expression Ex>
    auto constexpr negative( Ex const& ex ) noexcept
    {
        return make_unary_operator( []<Tensor Tsor>( Tsor const& tensor ) noexcept
                                    {
                                        better_assert( !has_nan( tensor ), "forward propagation for operator log:  tensor contains Nan!" );
                                        return -tensor;
                                    },
                                    []<Tensor Tsor>( Tsor const&, Tsor const&, Tsor const& grad ) noexcept
                                    {
                                        better_assert( !has_nan( grad ), "input gradient for operator negative contains NaN!" );
                                        return -grad;
                                    },
                                    "Negative"
                )( ex );
    };

    template <Expression Ex>
    auto constexpr operator - ( Ex const& ex ) noexcept
    {
        return negative( ex );
    }


    template <Expression Ex>
    auto constexpr inverse( Ex const& ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();
        return make_unary_operator( [forward_cache]<Tensor Tsor>( Tsor const& tensor ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( forward_cache );
                                        ans.resize( tensor.shape() );
                                        //for_each( tensor.begin(), tensor.end(), ans.begin(), [](auto const x, auto& y) { y = (x > 0.0) ?  (1.0/std::max(eps, x)) :  (1.0/std::min(-eps, x)); });
                                        for_each( tensor.begin(), tensor.end(), ans.begin(), [](auto const x, auto& y) { y = 1.0/x; } );
                                        return ans;
                                    },
                                    [backward_cache]<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor const& grad ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( backward_cache );
                                        ans.resize( input.shape() );
                                        //for_each( ans.begin(), ans.end(), grad.begin(), input.begin(), []( auto& x, auto y, auto z ){ x = - y / std::max(z*z, eps); } );
                                        for_each( ans.begin(), ans.end(), grad.begin(), input.begin(), []( auto& x, auto y, auto z ){ x = - y / (z*z); } );
                                        ans.resize( grad.shape() );
                                        return ans;
                                    },
                                    "Inverse"
                )( ex );
    };




    template< Expression Lhs_Expression, Expression Rhs_Expression >
    auto constexpr elementwise_product( Lhs_Expression const& lhs_ex, Rhs_Expression const& rhs_ex ) noexcept
    {
        return make_binary_operator( []<Tensor Tsor>( Tsor const& lhs_tensor, Tsor const& rhs_tensor ) noexcept
                                     {
                                        return elementwise_product( lhs_tensor, rhs_tensor );
                                     },
                                     []<Tensor Tsor>( Tsor const& lhs_input, Tsor const& rhs_input, Tsor const&, Tsor const grad ) noexcept
                                     {
                                        auto const& grad_fun = [&grad]( auto const& input, auto const& other_input )
                                        {
                                            Tsor ans = elementwise_product( grad, other_input );
                                            while( input.ndim() < ans.ndim() )
                                                ans = sum( ans, 0 );
                                            auto const& shape = input.shape();
                                            for ( auto axis :  range( input.ndim() ) )
                                                if ( shape[axis] == 1 )
                                                    ans = sum( ans, axis, true );
```

```
                                        return ans;
                                    };
                                    return std::make_tuple( grad_fun( lhs_input, rhs_input ), grad_fun( rhs_input, lhs_input ) );
                                },
                                "HadamardProduct"
                )( lhs_ex, rhs_ex );
    };

    template< Expression Lhs_Expression, Expression Rhs_Expression >
    auto constexpr elementwise_multiply( Lhs_Expression const& lhs_ex, Rhs_Expression const& rhs_ex ) noexcept
    {
        return elementwise_product( lhs_ex, rhs_ex );
    }

    template< Expression Lhs_Expression, Expression Rhs_Expression >
    auto constexpr hadamard_product( Lhs_Expression const& lhs_ex, Rhs_Expression const& rhs_ex ) noexcept
    {
        return elementwise_product( lhs_ex, rhs_ex );
    }


    template< Expression Lhs_Expression, Expression Rhs_Expression >
    auto constexpr divide( Lhs_Expression const& lhs_ex, Rhs_Expression const& rhs_ex ) noexcept
    {
        return elementwise_product( lhs_ex, inverse( rhs_ex ) );
    }

    template< Expression Lhs_Expression, Expression Rhs_Expression >
    auto constexpr operator / ( Lhs_Expression const& lhs_ex, Rhs_Expression const& rhs_ex ) noexcept
    {
        return divide( lhs_ex, rhs_ex );
    }




    template <Expression Ex>
    auto constexpr sum_reduce( Ex const& ex ) noexcept
    {
        return make_unary_operator( []<Tensor Tsor>( Tsor const& tsor ) noexcept
                                {
                                    better_assert( !has_nan( tsor ), "forward propagation for operator sum_reduce:  tensor contains Nan!" );
                                    return reduce_sum( tsor );
                                },
                                []<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor const& grad ) noexcept
                                {
                                    better_assert( !has_nan( grad ), "input gradient for operator sum_reduce contains NaN!" );
                                    better_assert( grad.size() == 1, "sum_reduce should only output one value" );
                                    Tsor ans = ones_like( input );
                                    ans *= grad[0];
                                    return ans;
                                },
                                "Sum",
                                []( std::vector<unsigned long> const& ) noexcept { return std::vector<unsigned long>{ {1,} }; }
                )( ex );
    }

    template <Expression Ex>
    auto constexpr reduce_sum( Ex const& ex ) noexcept
    {
        return sum_reduce( ex );
    }

    template <Expression Ex>
    auto constexpr mean_reduce( Ex const& ex ) noexcept
    {
        return make_unary_operator( []<Tensor Tsor>( Tsor const& tsor ) noexcept
                                {
                                    better_assert( !has_nan( tsor ), "forward propagation for operator mean:  tensor contains Nan!" );
                                    return reduce_mean( tsor );
                                },
                                []<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor const& grad ) noexcept
                                {
                                    better_assert( !has_nan( grad ), "input gradient for operator mean_reduce contains NaN!" );
                                    better_assert( grad.size() == 1, "mean_reduce should only output one value" );
```

```
                                Tsor ans = ones_like( input );
                                ans *= grad[0];
                                unsigned long const batch_size = (input.shape().size() == 1) ? 1 :  (*(input.shape().begin()));
                                ans /= static_cast<typename Tsor::value_type>(batch_size);
                                return ans;
                            },
                            "Mean",
                            []( std::vector<unsigned long> const& ) noexcept { return std::vector<unsigned long>{ {1,} }; }
                )( ex );
    }

    template <Expression Ex>
    auto constexpr reduce_mean( Ex const& ex ) noexcept
    {
        return mean_reduce( ex );
    }

    template <Expression Ex>
    auto constexpr mean( Ex const& ex ) noexcept
    {
        return mean_reduce( ex );
    }




    template< Expression Lhs_Expression, Expression Rhs_Expression >
    auto constexpr minus( Lhs_Expression const& lhs_ex, Rhs_Expression const& rhs_ex ) noexcept
    {
        if constexpr (is_value_v<Rhs_Expression>)
        {
            return negative( plus( negative(lhs_ex), rhs_ex ) );
        }
        else
        {
            return plus( lhs_ex, negative(rhs_ex) );
        }
    }

    template< Expression Lhs_Expression, Expression Rhs_Expression >
    auto constexpr operator - ( Lhs_Expression const& lhs_ex, Rhs_Expression const& rhs_ex ) noexcept
    {
        return minus( lhs_ex, rhs_ex );
    }


    template <Expression Ex>
    auto constexpr square( Ex const& ex ) noexcept
    {
        return make_unary_operator( []<Tensor Tsor>( Tsor const& tsor ) noexcept
                            {
                                better_assert( !has_nan( tsor ), "forward propagation for operator square:  tensor contains Nan!" );
                                Tsor ans = tsor.deep_copy();
                                std::for_each( ans.data(), ans.data() + ans.size(), []( auto & v ){ v *= v; } );
                                return ans;
                            },
                            []<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor const& grad ) noexcept
                            {
                                better_assert( !has_nan( grad ), "input gradient for operator square contains NaN!" );
                                Tsor ans = input.deep_copy();
                                ans *= grad;
                                ans *= typename Tsor::value_type{2};
                                return ans;
                            },
                            "Square"
                )( ex );
    }


    template <Expression Ex, Expression Ey>
    auto constexpr hypot( Ex const& ex, Ey const& ey ) noexcept
    {
        return sqrt( square(ex) + square(ey) );
    }





    template <typename Float> requires std::floating_point<Float>
    auto constexpr clip( Float lower, Float upper=std::numeric_limits<Float>::max() ) noexcept
```

```
    {
        return [lower, upper]<Expression Ex>( Ex const& ex ) noexcept
        {
            return make_unary_operator( [lower, upper]<Tensor Tsor>( Tsor const& tsor ) noexcept
                                        {
                                            better_assert( !has_nan( tsor ), "forward propagation for operator clip:  tensor contains Nan!" );
                                            Tsor ans = tsor.deep_copy();
                                            clip( ans, lower, upper );
                                            return ans;
                                        },
                                        [lower, upper]<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor const& grad ) noexcept
                                        {
                                            better_assert( !has_nan( grad ), "input gradient for operator clip contains NaN!" );
                                            const typename Tsor::value_type zero{0};
                                            Tsor ans = grad;
                                            for ( auto idx :  range( input.size() ) )
                                                ans[idx] = (input[idx] < lower) ?  zero :
                                                           (input[idx] > upper) ?  zero :
                                                           ans[idx];
                                            return ans;
                                        },
                                        "Clip"
                    )( ex );
        };
    }

    // include_batch_flag:
    //
    //  true:  considering the batch size at the first dim
    //      - for an input of (1, 3, 4), expecting an incoming expression of shape like [BS, 12, 1 1]
    //      - expected output of shape [BS, 1, 3, 4]
    //  false:  do not consider the batch size
    //      - for an input of (1, 3, 4), expecting an incoming expression of shape like [12, 1]
    //      - expected output of shape [1, 3, 4]
    auto inline reshape( std::vector<unsigned long> const& new_shape, bool include_batch_flag=true ) noexcept
    {
        return [new_shape, include_batch_flag]<Expression Ex>( Ex const& ex ) noexcept
        {
            return make_unary_operator
            (
                [new_shape, include_batch_flag]<Tensor Tsor>( Tsor const& tsor ) noexcept
                {
                    unsigned long const new_size = std::accumulate( new_shape.begin(), new_shape.end(), 1UL, []( auto x, auto y ){ return x*y; } );
                    unsigned long const total_size = tsor.size();
                    unsigned long const batch_size = total_size / new_size;

                    better_assert( batch_size * new_size == total_size, "size mismatch for reshape operator, expect ",  batch_size*new_size, " but total input size is ", total_size, ", where batch_size is ", batch_size );

                    if ( !include_batch_flag )
                    {
                        better_assert( batch_size == 1, "expecting batch size of 1 while not including batch, but got ", batch_size );
                        Tsor ans{tsor};
                        ans.reshape( new_shape );
                        return ans;
                    }

                    std::vector<unsigned long> batched_new_shape;
                    {
                        batched_new_shape.resize( 1 + new_shape.size() );
                        batched_new_shape[0] = batch_size;
                        std::copy( new_shape.begin(), new_shape.end(), batched_new_shape.begin()+1 );
                    }

                    Tsor ans{ tsor };
                    ans.reshape( batched_new_shape );
                    return ans;
                },
                []<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor const& grad ) noexcept
                {
                    Tsor ans{ grad };
                    ans.reshape( input.shape() );
                    return ans;
                },
                "Reshape",
                [new_shape, include_batch_flag]( std::vector<unsigned long> const& shape ) noexcept
                {

                    //debug_log( fmt::format("Calculating Reshape layer size include_batch_flag = {}", include_batch_flag) );
```

```
                    //debug_log( fmt::format("Calculating Reshape layer size with shape = {}", shape) );
                    //debug_log( fmt::format("Calculating Reshape layer size with new_shape = {}, ", new_shape) );

                    if ( include_batch_flag == false )
                        return new_shape;

                    unsigned long const new_size = std::accumulate( new_shape.begin(), new_shape.end(), 1UL, []( auto x, auto y ){ return x*y; } );
                    unsigned long const total_size = std::accumulate( shape.begin(), shape.end(), 1UL, []( unsigned long x, unsigned long y ){ return x*y; } );
                    unsigned long const batch_size = total_size / new_size;
                    std::vector<unsigned long> batched_new_shape;
                    {
                        batched_new_shape.resize( 1 + new_shape.size() );
                        batched_new_shape[0] = batch_size;
                        std::copy( new_shape.begin(), new_shape.end(), batched_new_shape.begin()+1 );
                    }
                    return batched_new_shape;

                    /*
std::vector<unsigned long> ans;
ans.resize( new_shape.size()+1 );
ans[0] = shape[0];
std::copy( new_shape.begin(), new_shape.end(), ans.begin()+1 );
return ans;
*/
                }
            )( ex );
        };
    }

    template <Expression Ex>
    auto constexpr flatten( Ex const& ex ) noexcept
    {
        return make_unary_operator
        (
            []<Tensor Tsor>( Tsor const& tsor ) noexcept
            {
                better_assert( tsor.ndim() > 1, "Expecting dimension of incoming tensor to be greater than 1, but got ", tsor.ndim() );
                unsigned long const batch_size = *(tsor.shape().begin());
                unsigned long const rem = tsor.size() / batch_size;
                Tsor ans = tsor;
                return ans.reshape( {batch_size, rem} );
            },
            []<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor const& grad ) noexcept
            {
                Tsor ans = grad;
                return ans.reshape( input.shape() );
            },
            "Flatten",
            []( std::vector<unsigned long> const& shape ) noexcept
            {
                unsigned long const total = std::accumulate( shape.begin()+1, shape.end(), 1, []( unsigned long x, unsigned long y ){ return x*y; } );
                return std::vector<unsigned long>{ {shape[0], total,} }; // the 1st dim is batch size
            }
        )( ex );
    }

    constexpr auto inline expand_dims( int axis=-1 ) noexcept
    {
        return [=]<Expression Ex>( Ex const& ex ) noexcept
        {
            return make_unary_operator
            (
                [=]<Tensor Tsor>( Tsor const& tsor ) noexcept
                {
                    Tsor ans = tsor;
                    std::vector<unsigned long> shape = ans.shape();
                    int const _axis = (axis == -1) ?  shape.size() :  axis;
                    shape.insert( shape.begin()+_axis, 1UL );
                    ans.reshape( shape );
                    return ans;
                },
                [=]<Tensor Tsor>( Tsor const& input, Tsor const& /*output*/, Tsor const& grad ) noexcept
                {
                    Tsor ans = grad;
                    ans.reshape( input.shape() );
                    return ans;
                },
                "ExpandDims",
                [axis]( std::vector<unsigned long> const& shape ) noexcept
                {
                    std::vector<unsigned long> ans = shape;
                    if ( axis == -1 ) axis = ans.size();
```

```
                    ans.insert( ans.begin()+axis, 1 );
                }
            )(ex);
        };
    }


    auto inline argmax( unsigned long axis=0 ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();

        return [=]<Expression Ex>( Ex const& ex ) noexcept
        {
            return make_unary_operator
            (
                [=]<Tensor Tsor>( Tsor const& input ) noexcept
                {
                    std::vector<unsigned long> const& shape = input.shape();
                    better_assert( axis < shape.size(), fmt::format("axis {} is greater than the dimension of the input tensor shape {}", axis, shape) );

                    // calculate the output tensor shape
                    std::vector<unsigned long> output_shape = shape;
                    std::copy( output_shape.begin()+axis+1, output_shape.end(), output_shape.begin()+axis );
                    output_shape.resize( output_shape.size() - 1 );
                    Tsor& ans = context_cast<Tsor>( forward_cache );
                    ans.resize( output_shape );

                    //  viewing the input tensor as a 3D tensor, and viewing the output tensor as a 2D tensor
                    unsigned long const bs = std::accumulate( shape.begin(), shape.begin()+axis, 1UL, []( unsigned long x, unsigned long y ){ return x*y; } );
                    unsigned long const row = shape[axis];
                    unsigned long const col = std::accumulate( shape.begin()+axis+1, shape.end(), 1UL, []( unsigned long x, unsigned long y ){ return x*y; } );
                    auto cube_input = view_3d{ input.data(), bs, row, col };
                    auto matrix_output = view_2d{ ans.data(), bs, col };

                    for ( auto _bs :  range( bs ) )
                        for ( auto _col :  range( col ) )
                        {
                            unsigned long mx_idx = 0;
                            auto mx = cube_input[_bs][0][_col];
                            for ( auto _row :  range( row ) )
                            {
                                if ( cube_input[_bs][_row][_col] > mx )
                                {
                                    mx = cube_input[_bs][_row][_col];
                                    mx_idx = _row;
                                }
                            }
                            matrix_output[_bs][_col] = mx_idx;
                        }
                    return ans;
                },
                [=]<Tensor Tsor>( Tsor const& input, Tsor const& /*output*/, Tsor const& /*grad*/ ) noexcept
                {
                    Tsor& back_ans = context_cast<Tsor>( backward_cache );
                    back_ans.resize( input.shape() );
                    for_each( back_ans.begin(), back_ans.end(), []( auto& v ){ v = 0.0; } ); // always return zero
                    return back_ans;
                },
                "Argmax",
                [axis]( std::vector<unsigned long> const& shape ) noexcept
                {
                    std::vector<unsigned long> ans = shape;
                    std::copy( ans.begin()+axis+1, ans.end(), ans.begin()+axis );
                    ans.resize( ans.size() - 1 );
                    return ans;
                }
            )(ex);
        };
    }


    auto inline argmin( unsigned long axis=0 ) noexcept
    {
        return [=]<Expression Ex>( Ex const& ex ) noexcept
        {
            return argmax(axis)( -ex );
        };
    }

```

```cpp


    template <Expression Ex>
    auto constexpr identity( Ex const& ex ) noexcept
    {
        return ex;
    }

    template< Expression Ex >
    auto transpose( Ex const& ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();
        return make_unary_operator
        (
            [forward_cache]<Tensor Tsor>( Tsor const& tsor ) noexcept
            {
                better_assert( tsor.ndim() == 2, "Expecting 2D tensor, but got dimensions ", tsor.ndim() );

                typedef typename Tsor::value_type value_type;

                std::vector<unsigned long> const shape = tsor.shape();
                auto const[row, col] = std::make_tuple( shape[0], shape[1] );
                view_2d<value_type> v_in{ tsor.data(), row, col };

                Tsor& ans = context_cast<Tsor>( forward_cache );
                ans.resize( {col, row} );
                view_2d<value_type> v_out{ ans.data(), col, row };

                for ( auto r :  range( row ) )
                    for ( auto c :  range( col ) )
                        v_out[c][r] = v_in[r][c];

                return ans;
            },
            [backward_cache]<Tensor Tsor>( Tsor const&, Tsor const&, Tsor const& grad ) noexcept
            {
                typedef typename Tsor::value_type value_type;

                std::vector<unsigned long> const shape = grad.shape();
                auto const[row, col] = std::make_tuple( shape[0], shape[1] );
                view_2d<value_type> v_in{ grad.data(), row, col };

                Tsor& back_ans = context_cast<Tsor>( backward_cache );
                back_ans.resize( {col, row} );

                view_2d<value_type> v_out{ back_ans.data(), col, row };

                for ( auto r :  range( row ) )
                    for ( auto c :  range( col ) )
                        v_out[c][r] = v_in[r][c];

                return back_ans;
            },
            "Transpose",
            []( std::vector<unsigned long> const shape ) noexcept
            {
                better_assert( shape.size() == 2, fmt::format( "expecting shape size of 2, but got {}", shape.size() ) );
                return std::vector<unsigned long>{ {shape[1], shape[0]} };
            }
        )( ex );
    }

    auto inline img2col( unsigned long const row_kernel, unsigned long col_kernel=-1,
                         unsigned long const row_padding=0, unsigned long col_padding=0,
                         unsigned long const row_stride=1, unsigned long const col_stride=1,
                         unsigned long const row_dilation=1, unsigned long const col_dilation=1 ) noexcept
    {
        if ( col_kernel == (unsigned long)-1 ) col_kernel = row_kernel;

        std::shared_ptr<std::vector<std::uint32_t» s_index_record = std::make_shared<std::vector<std::uint32_t»(); // col_img[idx] = img[index_record[idx]]  -- (-1) for zero padding

        auto img2col_forward = [s_index_record]<Tensor Tsor>
        (
            Tsor const& input_img, Tsor& output_col_mat,
            unsigned long kernel_row, unsigned long kernel_col,
            unsigned long padding_row, unsigned long padding_col,
            unsigned long stride_row, unsigned long stride_col,
            unsigned long dilation_row, unsigned long dilation_col
        ) noexcept
        {
            typedef typename Tsor::value_type value_type;
```

```
            std::vector<std::uint32_t>& index_record = *s_index_record; //32 bit should be enough for memory address offeset

            std::vector<unsigned long> input_shape = input_img.shape();
            better_assert( input_shape.size() == 4, "Expecting a 4D tensor." );
            auto const [BS, R, C, CH] = std::make_tuple( input_shape[0], input_shape[1], input_shape[2], input_shape[3] );

            unsigned long const output_row = ( R + 2 * padding_row - ( dilation_row * (kernel_row - 1) + 1 ) ) / stride_row + 1;
            unsigned long const output_col = ( C + 2 * padding_col - ( dilation_col * (kernel_col - 1) + 1 ) ) / stride_col + 1;
            unsigned long const output_column_matrix_row = kernel_row * kernel_col * CH;
            unsigned long const output_column_matrix_col = BS * output_row * output_col;

            output_col_mat.resize( {output_column_matrix_row, output_column_matrix_col} );

            if ( index_record.size() != output_column_matrix_row * output_column_matrix_col ) // first-run?
            {
                index_record.resize( output_column_matrix_row * output_column_matrix_col );

                for ( auto bs :  range( BS ) )
                {
                    std::int64_t const col_offset = bs * output_row * output_col * kernel_row * kernel_col * CH;
                    std::int64_t const im_offset = bs * R * C * CH;
                    for ( auto c :  range( CH * kernel_row * kernel_col ) )
                    {
                        std::int64_t const w_offset = c % kernel_col;
                        std::int64_t const h_offset = ( c / kernel_col ) % kernel_row;
                        std::int64_t const c_im = c / ( kernel_col * kernel_row );

                        for ( auto h :  range( output_row ) )
                        {
                            std::int64_t const im_row_idx = h * stride_row - padding_row + h_offset * dilation_row;
                            for ( auto w :  range( output_col ) )
                            {
                                std::int64_t const im_col_idx = w * stride_col - padding_col + w_offset * dilation_col;
                                std::int64_t const im_idx = im_offset+( im_row_idx * C + im_col_idx ) * CH + c_im;
                                std::int64_t const col_idx = col_offset+( c * output_row + h ) * output_col + w;
                                index_record[col_idx] = static_cast<std::uint32_t>((im_row_idx<0 || im_row_idx>=static_cast<std::int64_t>(R) || im_col_idx<0 || im_col_idx>=static_cast<std::int64_t>(C)) ? 0xffffffff :  im_idx);
                            }
                        }
                    }
                }
                // re-arrange [bs, new_R, new_C] --> [new_R, new_c*bs]
                {
                    std::vector<std::uint32_t> re_arranged_index;
                    re_arranged_index.resize( index_record.size() );

                    view_3d<std::uint32_t> re_arranged_mat{ re_arranged_index.data(), output_column_matrix_row, BS, output_row*output_col };
                    view_3d<std::uint32_t> index_record_mat{ index_record.data(), BS, output_column_matrix_row, output_row*output_col };

                    for ( auto bs :  range( BS ) )
                        for ( auto r :  range( output_column_matrix_row ) )
                            for ( auto c :  range( output_row*output_col ) )
                                re_arranged_mat[r][bs][c] = index_record_mat[bs][r][c];
                    // overwrite index record
                    std::copy( re_arranged_index.begin(), re_arranged_index.end(), index_record.begin() );
                }
            }

            // fill-in
            for ( auto idx :  range( output_col_mat.size() ) )
            {
                auto const index = index_record[idx];
                output_col_mat[idx] = (index == 0xffffffff) ?  value_type{0} :  input_img[index];
            }
        };

        auto img2col_backward = [s_index_record]<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor const& grad, Tsor& ans ) noexcept
        {
            typedef typename Tsor::value_type value_type;
            ans.resize( input.shape() );
            std::fill( ans.begin(), ans.end(), value_type{0} );

```

```
            std::vector<std::uint32_t>& index_record = *s_index_record; //32 bit should be enough for memory address offeset
            for ( auto idx :  range( grad.size() ) )
            {
                auto const index = index_record[idx];
                if ( index != 0xffffffff )
                    ans[index] += grad[idx];
            }
        };

        std::shared_ptr<std::any> output_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> back_grad_cache = std::make_shared<std::any>();

        return [row_kernel, col_kernel, row_padding, col_padding, row_stride, col_stride, row_dilation, col_dilation, img2col_forward, img2col_backward, output_cache, back_grad_cache]<Expression Ex>( Ex const& ex ) noexcept
        {
            return make_unary_operator
            (
                [=]<Tensor Tsor>( Tsor const & tsor ) noexcept
                {
                    Tsor& output = context_cast<Tsor>( output_cache );
                    img2col_forward( tsor, output, row_kernel, col_kernel, row_padding, col_padding, row_stride, col_stride, row_dilation, col_dilation );
                    return Tsor{output};
                },
                [=]<Tensor Tsor>( Tsor const& input, Tsor const& output, Tsor const& grad ) noexcept
                {
                    Tsor& back_grad = context_cast<Tsor>( back_grad_cache );
                    img2col_backward( input, output, grad, back_grad );
                    return Tsor{back_grad};
                },
                "Img2Col",
                [=]( std::vector<unsigned long> const& shape ) noexcept
                {
                    better_assert( shape.size() == 4, fmt::format("Expecting a 4D tensor, but got {}.", shape.size()) );
                    auto const [BS, R, C, CH] = std::make_tuple( shape[0], shape[1], shape[2], shape[3] );

                    unsigned long const output_row = ( R + 2 * row_padding - ( row_dilation * (row_kernel - 1) + 1 ) ) / row_stride + 1;
                    unsigned long const output_col = ( C + 2 * col_padding - ( col_dilation * (col_kernel - 1) + 1 ) ) / col_stride + 1;
                    unsigned long const output_column_matrix_row = row_kernel * col_kernel * CH;
                    unsigned long const output_column_matrix_col = BS * output_row * output_col;
                    return std::vector<unsigned long>{ {output_column_matrix_row, output_column_matrix_col} };
                }
            )( ex );
        };
    }

    auto inline conv2d
    (
        unsigned long row_input, unsigned long col_input,
        unsigned long const row_stride=1, unsigned long const col_stride=1,
        unsigned long const row_dilation=1, unsigned long const col_dilation=1,
        std::string const& padding="valid"
    ) noexcept
    {
        // lhs_ex is for one 4D tensor of [BS, R, C, CH]
        // rhs_ex is for NC 4D filter of [1, r, c, CH], thus the shape is [NC, r, c, CH]
        // the output tensor is of shape [BS, .., .., NC]
        //
        // Note:  the rhs expression is fixed as a variable, as we need to extract the kernel shape from it
        //
        //return [row_input, col_input, row_stride, col_stride, row_dilation, col_dilation, padding ]<Expression Ex, Variable Va>( Ex const& lhs_ex, Va const& rhs_ex ) noexcept
        return [row_input, col_input, row_stride, col_stride, row_dilation, col_dilation, padding ]<Expression Ex, Expression Ey>( Ex const& lhs_ex, Ey const& rhs_ex ) noexcept
        {
            std::vector<unsigned long> const& shape = rhs_ex.shape();
            better_assert( shape.size() == 4 );
            auto const[new_channel, row_kernel, col_kernel, channel] = std::make_tuple( shape[0], shape[1], shape[2], shape[3] );
            //TODO: optimization in case of small kernels of (1, 1), (3, 3)
            unsigned long row_padding = 0;
            unsigned long col_padding = 0;
            if ( padding == "same" )
            {
                unsigned long const row_padding_total = (row_kernel + (row_kernel - 1) * (row_dilation - 1) - row_stride);
                better_assert( !(row_padding_total & 0x1), "Expecting total row padding to be even, but got ", row_padding_total, " With row input ", row_input, " and row_stride ", row_stride );
                unsigned long const col_padding_total = (col_kernel + (col_kernel - 1) * (col_dilation
```

```
- 1) - col_stride);
                better_assert( !(col_padding_total & 0x1), "Expecting total col padding to be even, but
got ", col_padding_total );
                row_padding = ((row_kernel&1)+row_padding_total) » 1;
                col_padding = ((col_kernel&1)+col_padding_total) » 1;
            }

            unsigned long const row_output = ( row_input + 2 * row_padding - ( row_dilation *
(row_kernel - 1) + 1 ) ) / row_stride + 1;
            unsigned long const col_output = ( col_input + 2 * row_padding - ( col_dilation *
(col_kernel - 1) + 1 ) ) / col_stride + 1;

            auto lhs_ex_as_col = img2col(row_kernel, col_kernel, row_padding, col_padding, row_stride,
col_stride, row_dilation, col_dilation)( lhs_ex ); // [BS, R, C, CH] ==> [r*c*CH, BS*new_row*new_col]

            auto rhs_ex_flatten = reshape({row_kernel*col_kernel*channel,})( rhs_ex ); // [NC, r, c,
CH] ==> [NC, r*c*CH]

            auto flatten_output = rhs_ex_flatten * lhs_ex_as_col; // [NC, BS * new_row * new_col]

            auto tr_output = transpose( flatten_output ); // [BS*new_row*new_col, NC]

            auto ans = reshape({row_output, col_output, new_channel})( tr_output );

            return ans;
        };
    }


    auto inline general_conv2d
    (
        unsigned long const row_stride=1, unsigned long const col_stride=1,
        unsigned long const row_dilation=1, unsigned long const col_dilation=1,
        std::string const& padding="valid"
    ) noexcept
    {
        // lhs_ex is for one 4D tensor of [BS, R, C, CH]
        // rhs_ex is for NC 4D filter of [1, r, c, CH], thus the shape is [NC, r, c, CH]
        // the output tensor is of shape [BS, .., .., NC]
        //
        // Note:  the rhs expression is fixed as a variable, as we need to extract the kernel shape
from it
        //
        return [ row_stride, col_stride, row_dilation, col_dilation, padding ]<Expression Ex,
Expression Ey>( Ex const& lhs_ex, Ey const& rhs_ex ) noexcept
        {
            auto const& lhs_shape = lhs_ex.shape();
            better_assert( lhs_shape.size() == 4, fmt::format( "expecting lhs_shape size of 4, but got
{}", lhs_shape.size() ) );
            auto [_bs, row_input, col_input, _ch] = std::make_tuple( lhs_shape[0], lhs_shape[1],
lhs_shape[2], lhs_shape[3] );

            std::vector<unsigned long> const& shape = rhs_ex.shape();
            better_assert( shape.size() == 4 );
            auto const[new_channel, row_kernel, col_kernel, channel] = std::make_tuple( shape[0],
shape[1], shape[2], shape[3] );
            unsigned long row_padding = 0;
            unsigned long col_padding = 0;
            if ( padding == "same" )
            {
                unsigned long const row_padding_total = (row_kernel + (row_kernel - 1) * (row_dilation
- 1) - row_stride);
                unsigned long const col_padding_total = (col_kernel + (col_kernel - 1) * (col_dilation
- 1) - col_stride);
                row_padding = ((row_kernel&1)+row_padding_total) » 1;
                col_padding = ((col_kernel&1)+col_padding_total) » 1;
            }

            unsigned long const row_output = ( row_input + 2 * row_padding - ( row_dilation *
(row_kernel - 1) + 1 ) ) / row_stride + 1;
            unsigned long const col_output = ( col_input + 2 * row_padding - ( col_dilation *
(col_kernel - 1) + 1 ) ) / col_stride + 1;

            auto lhs_ex_as_col = img2col(row_kernel, col_kernel, row_padding, col_padding, row_stride,
col_stride, row_dilation, col_dilation)( lhs_ex ); // [BS, R, C, CH] ==> [r*c*CH, BS*new_row*new_col]

            auto rhs_ex_flatten = reshape({row_kernel*col_kernel*channel,})( rhs_ex ); // [NC, r, c,
CH] ==> [NC, r*c*CH]

            auto flatten_output = rhs_ex_flatten * lhs_ex_as_col; // [NC, BS * new_row * new_col]

            auto tr_output = transpose( flatten_output ); // [BS*new_row*new_col, NC]

            auto ans = reshape({row_output, col_output, new_channel})( tr_output );

            return ans;
        };
```

```
    }

    template< typename T > requires std::floating_point<T>
    inline auto drop_out( T const factor ) noexcept
    {
        better_assert( factor < T{1}, "Expecting drop out rate less than 1, but got factor = ", factor );
        better_assert( factor > T{0}, "Expecting drop out rate greater than 0, but got factor = ", factor );

        std::shared_ptr<std::any> mask = std::make_shared<std::any>();
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();

        return [factor, mask, forward_cache, backward_cache]<Expression Ex>( Ex const& ex ) noexcept
        {
            return make_unary_operator
            (
                [factor, mask, forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                {
                    typedef typename Tsor::value_type value_type;

                    if ( learning_phase == 0 ) // defined in 'config.hpp'
                        return input;

                    std::any& mask_ = *mask;
                    // first run, initialize mask
                    if ( !mask_.has_value() )
                    {
                        Tsor const random_tensor = random<value_type>( input.shape() );
                        Tsor mask__{ input.shape() };
                        for ( auto idx :  range( input.size() ) )
                            if ( random_tensor[ idx ] > factor )
                                mask__[ idx ] = 1;
                        mask_ = mask__; // initialize
                    }

                    Tsor& mask__ = std::any_cast<Tsor&>( mask_ );

                    Tsor& ans = context_cast<Tsor>( forward_cache );
                    ans.deep_copy( input );

                    for ( auto idx :  range( input.size() ) )
                        ans[idx] *= mask__[idx] / (value_type{1} - factor);
                    return ans;
                },
                [mask, backward_cache]<Tensor Tsor>( Tsor const&, Tsor const&, Tsor const& grad ) noexcept
                {
                    if ( learning_phase == 0 ) // defined in 'config.hpp'
                        return grad;

                    Tsor& mask__ = std::any_cast<Tsor&>( *mask );

                    Tsor& ans = context_cast<Tsor>( backward_cache );
                    ans.deep_copy( grad );

                    for ( auto idx :  range( grad.size() ) )
                        ans[idx] *= mask__[idx];
                    return ans;
                },
                "Dropout"
            )( ex );
        };
    }

    template< typename T > requires std::floating_point<T>
    inline auto dropout( T const factor ) noexcept
    {
        return drop_out( factor );
    }


    namespace
    {

        struct max_pooling_2d_context
        {

            auto make_forward() const noexcept
            {
```

```
                return  []( unsigned long stride, std::shared_ptr<std::any> mask, std::shared_ptr<std::any> forward_cache ) noexcept
                {
                    return [=]<Tensor Tsor>( Tsor const& input ) noexcept
                    {
                        typedef typename Tsor::value_type value_type;
                        better_assert( input.ndim() == 4, "Expecting a 4D tensor, but got ", input.ndim() );

                        Tsor& mask__ = context_cast<Tsor>( mask );
                        mask__.resize( input.shape() );


                        std::vector<unsigned long> shape = input.shape();
                        auto const[batch_size, row, col, channel] = std::make_tuple(shape[0], shape[1], shape[2], shape[3]);
                        Tsor input_ = input;
                        view_4d<value_type> ts{ input_.data(), batch_size, row, col, channel };
                        view_4d<value_type> tm{ mask__.data(), batch_size, row, col, channel };

                        Tsor& ans = context_cast<Tsor>( forward_cache );
                        ans.resize( {batch_size, row/stride, col/stride, channel} );

                        view_4d<value_type> t1{ ans.data(), batch_size, row/stride, col/stride, channel };

                        for ( auto bs :  range(batch_size) )
                            for ( auto r :  range(row/stride) ) // row for t1
                                for ( auto c :  range(col/stride) ) // col for t1
                                    for ( auto ch :  range(channel) )
                                    {
                                        unsigned long current_row_max = r * stride;
                                        unsigned long current_col_max = c * stride;
                                        for ( auto _r :  range( (r*stride), ((r*stride)+stride) ) ) // row for ts
                                            for ( auto _c :  range( (c*stride), ((c*stride)+stride) ) ) // col for ts
                                            {
                                                if ( ts[bs][_r][_c][ch] > ts[bs][current_row_max][current_col_max][ch] )
                                                {
                                                    current_row_max = _r;
                                                    current_col_max = _c;
                                                }
                                            }
                                        tm[bs][current_row_max][current_col_max][ch] = 1.0; //mark as max
                                        t1[bs][r][c][ch] = ts[bs][current_row_max][current_col_max][ch]; // update value
                                    }
                        return ans;
                    };
                };
            }

            auto make_backward() const noexcept
            {
                return []( unsigned long stride, std::shared_ptr<std::any> mask, std::shared_ptr<std::any> backward_cache ) noexcept
                {
                    return [=]<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor const& grad ) noexcept
                    {
                        typedef typename Tsor::value_type value_type;
                        std::vector<unsigned long> const& shape = input.shape();
                        auto const[batch_size, row, col, channel] = std::make_tuple(shape[0], shape[1], shape[2], shape[3]);

                        Tsor& mask__ = std::any_cast<Tsor&>( *mask );
                        view_4d<value_type> tm{ mask__.data(), batch_size, row, col, channel };

                        Tsor& ans = context_cast<Tsor>( backward_cache );
                        ans.resize( input.shape() );

                        view_4d<value_type> ta{ ans.data(), batch_size, row, col, channel };

                        Tsor grad_ = grad;
                        view_4d<value_type> tg{ grad_.data(), batch_size, row/stride, col/stride, channel };

                        for ( auto bs :  range( batch_size ) )
                            for ( auto r :  range( row ) )
                                for ( auto c :  range( col ) )
                                    for ( auto ch :  range( channel ) )
                                        if ( std::abs(tm[bs][r][c][ch] - 1.0) < 1.0e-5 )
                                            ta[bs][r][c][ch] = tg[bs][r/stride][c/stride][ch];
                        return ans;
```

```
                    };
                };
            }

        }; // max_pooling_2d_context

    } // anonymous namespace


    // comment:  maybe using function 'reduce' to reduce the cod complexity?  at a price of performance?
    inline auto max_pooling_2d( unsigned long stride ) noexcept
    {
        better_assert( stride > 1, "Expecting max_pooling_2d stride greater than 1, but got ", stride );

        std::shared_ptr<std::any> mask = std::make_shared<std::any>();
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();

        return [stride, mask, forward_cache, backward_cache]<Expression Ex>( Ex const& ex ) noexcept
        {
            return make_unary_operator
            (
                max_pooling_2d_context{}.make_forward()( stride, mask, forward_cache ),
                max_pooling_2d_context{}.make_backward()( stride, mask, backward_cache ),
                "MaxPooling2D",
                [=]( std::vector<unsigned long> const& shape ) noexcept
                {
                    better_assert( shape.size()==4, fmt::format( "expecting shape size of 4, but got {}", shape.size() ) );
                    return std::vector<unsigned long>{ {shape[0], shape[1]/stride, shape[2]/stride, shape[3]} };
                }
            )( ex );
        };
    }

    inline auto average_pooling_2d( unsigned long stride ) noexcept
    {
        better_assert( stride > 1, "Expecting average_pooling_2d stride greater than 1, but got ", stride );

        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();

        return [stride, forward_cache, backward_cache]<Expression Ex>( Ex const& ex ) noexcept
        {
            return make_unary_operator
            (
                [stride, forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept // [BS, R, C, CH] --> [BS, R/s, C/s, CH]
                {
                    typedef typename Tsor::value_type value_type;
                    better_assert( input.ndim() == 4, "Expecting a 4D tensor, but got ", input.ndim() );

                    std::vector<unsigned long> shape = input.shape();
                    auto const[batch_size, row, col, channel] = std::make_tuple(shape[0], shape[1], shape[2], shape[3]);
                    Tsor input_ = input;
                    view_4d<value_type> ts{ input_.data(), batch_size, row, col, channel };

                    Tsor& ans = context_cast<Tsor>( forward_cache );
                    ans.resize( {batch_size, row/stride, col/stride, channel} );
                    std::fill( ans.begin(), ans.end(), value_type{0} );

                    view_4d<value_type> t1{ ans.data(), batch_size, row/stride, col/stride, channel };

                    value_type const factor = value_type{1} / static_cast<value_type>(stride*stride);
                    for ( auto bs :  range(batch_size) )
                        for ( auto r :  range(row/stride) ) // row for t1
                            for ( auto c :  range(col/stride) ) // col for t1
                                for ( auto ch :  range(channel) )
                                    for ( auto _r :  range( (r*stride), ((r*stride)+stride) ) ) // row for ts
                                        for ( auto _c :  range( (c*stride), ((c*stride)+stride) ) ) // col for ts
                                            t1[bs][r][c][ch] += ts[bs][_r][_c][ch] * factor;
                    return ans;
                },
                [stride, backward_cache]<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor const& grad ) noexcept
                {
                    typedef typename Tsor::value_type value_type;
                    std::vector<unsigned long> const& shape = input.shape();
                    auto const[batch_size, row, col, channel] = std::make_tuple(shape[0], shape[1],
```

```
shape[2], shape[3]);

                    Tsor& ans = context_cast<Tsor>( backward_cache );
                    ans.resize( input.shape() );

                    view_4d<value_type> ta{ ans.data(), batch_size, row, col, channel };

                    Tsor grad_ = grad;
                    view_4d<value_type> tg{ grad_.data(), batch_size, row/stride, col/stride, channel };

                    value_type const factor = value_type{1} / static_cast<value_type>(stride*stride);
                    for ( auto bs :  range( batch_size ) )
                        for ( auto r :  range( row ) )
                            for ( auto c :  range( col ) )
                                for ( auto ch :  range( channel ) )
                                    ta[bs][r][c][ch] = factor * tg[bs][r/stride][c/stride][ch];
                    return ans;
                },
                "AveragePooling2D",
                [=]( std::vector<unsigned long> const& shape ) noexcept
                {
                    better_assert( shape.size()==4, fmt::format( "expecting shape size of 4, but got {}", shape.size() ) );
                    return std::vector<unsigned long>{ {shape[0], shape[1]/stride, shape[2]/stride, shape[3]} };
                }
            )( ex );
        };
    }

    namespace
    {
        struct up_sampling_2d_context
        {
            auto make_forward() const noexcept
            {
                return []( unsigned long stride, std::shared_ptr<std::any> forward_cache ) noexcept
                {
                    return [=]<Tensor Tsor>( Tsor const& input ) noexcept
                    {
                        typedef typename Tsor::value_type value_type;
                        better_assert( input.ndim() == 4, "Expecting a 4D tensor, but got ", input.ndim() );

                        std::vector<unsigned long> shape = input.shape();
                        auto const[batch_size, row, col, channel] = std::make_tuple(shape[0], shape[1], shape[2], shape[3]);
                        Tsor input_ = input;
                        view_4d<value_type> ts{ input_.data(), batch_size, row, col, channel };

                        Tsor& ans = context_cast<Tsor>( forward_cache );
                        ans.resize( {batch_size, row*stride, col*stride, channel} );
                        std::fill( ans.begin(), ans.end(), value_type{0} );

                        view_4d<value_type> t1{ ans.data(), batch_size, row*stride, col*stride, channel };

                        for ( auto bs :  range(batch_size) )
                            for ( auto r :  range(row) ) // row for ts
                                for ( auto c :  range(col) ) // col for ts
                                    for ( auto ch :  range(channel) )
                                        for ( auto _r :  range( (r*stride), ((r*stride)+stride) ) ) // row for t1
                                            for ( auto _c :  range( (c*stride), ((c*stride)+stride) ) ) // col for t1
                                                t1[bs][_r][_c][ch] = ts[bs][r][c][ch];
                        return ans;
                    };
                };
            }

            auto make_backward() const noexcept
            {
                return []( unsigned long stride, std::shared_ptr<std::any> backward_cache ) noexcept
                {
                    return [=]<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor const& grad ) noexcept
                    {
                        typedef typename Tsor::value_type value_type;
                        std::vector<unsigned long> const& shape = input.shape();
                        auto const[batch_size, row, col, channel] = std::make_tuple(shape[0], shape[1], shape[2], shape[3]);

                        Tsor& ans = context_cast<Tsor>( backward_cache );
                        ans.resize( input.shape() );
                        std::fill( ans.begin(), ans.end(), value_type{0} );
```

```

                        view_4d<value_type> ta{ ans.data(), batch_size, row, col, channel };

                        Tsor grad_ = grad;
                        view_4d<value_type> tg{ grad_.data(), batch_size, row*stride, col*stride, channel };

                        for ( auto bs :  range( batch_size ) )
                            for ( auto r :  range( row ) )
                                for ( auto c :  range( col ) )
                                    for ( auto ch :  range( channel ) )
                                        for ( auto _r :  range( (r*stride), ((r*stride)+stride) ) ) // row for tg
                                            for ( auto _c :  range( (c*stride), ((c*stride)+stride) ) ) // col for tg
                                                ta[bs][r][c][ch] += tg[bs][_r][_c][ch];
                        return ans;
                    };
                };
            }
        }; // up_sampling_2d_context

    } // anonymous namespace

    inline auto up_sampling_2d( unsigned long stride ) noexcept
    {
        better_assert( stride > 1, "Expecting up_sampling_pooling_2d stride greater than 1, but got ", stride );

        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();

        return [stride, forward_cache, backward_cache]<Expression Ex>( Ex const& ex ) noexcept
        {
            return make_unary_operator
            (
                up_sampling_2d_context{}.make_forward()( stride, forward_cache ),
                up_sampling_2d_context{}.make_backward()( stride, backward_cache ),
                "UpSampling2D",
                [=]( std::vector<unsigned long> const& shape ) noexcept
                {
                    better_assert( shape.size()==4, fmt::format( "expecting shape size of 4, but got {}", shape.size() ) );
                    return std::vector<unsigned long>{ {shape[0], shape[1]*stride, shape[2]*stride, shape[3]} };
                }
            )( ex );
        };
    }

    // an alias name
    inline auto upsampling_2d( unsigned long stride ) noexcept
    {
        return up_sampling_2d( stride );
    }



    template< typename T=double > requires std::floating_point<T>
    inline auto normalization_batch( T const momentum=0.98 ) noexcept
    {
        std::shared_ptr<std::any> global_average_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> global_variance_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> average_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> variance_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();

        return [=]<Expression Ex>( Ex const& ex ) noexcept
        {
            return make_unary_operator
            (
                [=]<Tensor Tsor>( Tsor const& input ) noexcept
                {
                    better_assert( input.ndim() > 1, "normalization_batch requires input dimension at least 2, got ", input.ndim() );

                    typedef typename Tsor::value_type value_type;
                    //typedef typename Tsor::allocator allocator;

                    std::vector<unsigned long> const& shape = input.shape();
                    unsigned long const channels = *(shape.rbegin());
                    unsigned long const rest_dims = input.size() / channels;

                    view_2d<value_type> input_{ input.data(), rest_dims, channels };

                    // case of prediction phase, in this phase, the batch size could be 1, and it is
```

```
not possible to calculate the variance
                if ( learning_phase == 0 ) // defined in 'config.hpp'
                {
                    // fix for the special case when prediction is executed before the training, typically in a GAN
                    Tsor& global_average_test = context_cast<Tsor>( global_average_cache );
                    if ( global_average_test.empty() )
                        return input;

                    // normal case.  i.e., the global_average_cache and global_variance_cache are not empty
                    Tsor& global_average = context_extract<Tsor>( global_average_cache );
                    Tsor& global_variance = context_extract<Tsor>( global_variance_cache );

                    Tsor& ans = context_cast<Tsor>( forward_cache, zeros_like( input ) );
                    ans.resize( input.shape() ); // well, the batch sizes for training and for prediction are not necessarily same

                    view_2d<value_type> ans_{ ans.data(), rest_dims, channels };
                    {
                        for ( auto r :  range( rest_dims ) )
                            for ( auto c :  range( channels ) )
                                ans_[r][c] = (input_[r][c] - global_average[c]) / std::sqrt( global_variance[c] + eps );
                    }
                    return ans;
                }

                //calculate average along the last channel
                Tsor& average = context_cast<Tsor>( average_cache );
                {
                    average.resize( {channels, } );
                    std::fill( average.begin(), average.end(), value_type{0} );

                    for ( auto idx :  range( rest_dims ) )
                        for ( auto jdx :  range( channels ) )
                            average[jdx] += input_[idx][jdx];

                    average /= static_cast<value_type>(rest_dims);
                }

                //calculate Variance along the last channel
                Tsor& variance = context_cast<Tsor>( variance_cache );
                {
                    variance.resize( {channels,} );
                    std::fill( variance.begin(), variance.end(), value_type{0} );
                    for ( auto idx :  range( rest_dims ) )
                        for ( auto jdx :  range( channels ) )
                            variance[jdx] += std::pow( input_[idx][jdx] - average[jdx], 2 );

                    variance /= static_cast<value_type>( rest_dims );
                }


                Tsor& ans = context_cast<Tsor>( forward_cache );
                ans.resize( input.shape() ); // the batch sizes for training and for prediction are not necessarily same
                view_2d<value_type> ans_{ ans.data(), rest_dims, channels };
                {
                    for ( auto idx :  range( rest_dims ) )
                        for ( auto jdx :  range( channels ) )
                            ans_[idx][jdx] = ( input_[idx][jdx] - average[jdx] ) / std::sqrt( variance[jdx] + eps );
                }

                // update global average and global variance
                {
                    Tsor& global_average = context_cast<Tsor>( global_average_cache, zeros_like( average ) );
                    // Note:  No obvious different is observed between initializing global_variance to zeros and to ones with MNIST example:
                    //        initializing global_variance to zeros, after 10 epochs mnist gives an error of 0.026
                    //        initializing global_variance to ones, after 10 epochs mnist gives an error of 0.028
                    Tsor& global_variance = context_cast<Tsor>( global_variance_cache, zeros_like( variance ) );
                    //Tsor& global_variance = context_cast<Tsor>( global_variance_cache, ones_like( variance ) );
                    for ( auto idx :  range( global_average.size() ) )
                    {
                        global_average[idx] = global_average[idx] * momentum + average[idx] * ( 1.0 - momentum );
                        global_variance[idx] = global_variance[idx] * momentum + variance[idx] * ( 1.0 - momentum );
                    }
                }
```

```
                    return ans;
                },

                [=]<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor const& grad ) noexcept
                {
                    typedef typename Tsor::value_type value_type;
                    Tsor& variance = context_extract<Tsor>( variance_cache );

                    std::vector<unsigned long> const& shape = input.shape();
                    unsigned long const channels = *(shape.rbegin());
                    unsigned long const rest_dims = input.size() / channels;

                    Tsor& ans = context_cast<Tsor>( backward_cache, zeros_like( input ) );
                    view_2d<value_type> ans_{ans.data(), rest_dims, channels };
                    view_2d<value_type> grad_{grad.data(), rest_dims, channels };
                    for ( auto r :  range( rest_dims ) )
                        for ( auto c :  range( channels ) )
                            ans_[r][c] = grad_[r][c] / std::sqrt( variance[c] + eps );
                    return ans;
                },
                "Normalization"
            )( ex );
        };
    }



    template< typename T > requires std::floating_point<T>
    inline auto batch_normalization( T const momentum=0.98 ) noexcept
    {
        return [=]<Expression Ex, Variable Va>( Ex const& ex, Va const& gamma, Va const& beta ) noexcept
        {
            return elementwise_product( normalization_batch(momentum)(ex), gamma ) + beta; // multiply and sum along the batch:  normalization is of shape [BS, R, C, CH], gamma/beta are of shape [R, C, CH]
        };
    }



    //
    //  example:
    //
    //      variable<tensor<float» a {...  };
    //      variable<tensor<float» b {...  };
    //      auto cab = concatenate( a, b )();
    //
    template< Expression Lhs_Expression, Expression Rhs_Expression >
    auto constexpr concatenate( Lhs_Expression const& lhs_ex, Rhs_Expression const& rhs_ex ) noexcept
    {
        return [&]( unsigned long axe = -1 ) noexcept
        {
            return make_binary_operator
            (
                [axe]<Tensor Tsor>( Tsor const& lhs_tensor, Tsor const& rhs_tensor ) noexcept
                {
                    return concatenate( lhs_tensor, rhs_tensor, axe );
                },
                [axe]<Tensor Tsor>( Tsor const& lhs_input, Tsor const& rhs_input, Tsor const&, Tsor const grad ) noexcept
                {
                    typedef typename Tsor::value_type value_type;

                    Tsor l_ans{ lhs_input.shape() };
                    Tsor r_ans{ rhs_input.shape() };
                    better_assert(  l_ans.size() + r_ans.size() == grad.size(), "size mismatch:  lhs size is ", l_ans.size(), " rhs size is ", r_ans.size(), " and grad size is ", grad.size(),
                                        " with lhs dim is ", l_ans.ndim(), "  and rhs dim is ", r_ans.ndim() );

                    // 2D view of grad
                    unsigned long const ax = (axe == (unsigned long)(-1)) ?  grad.ndim()-1 :  axe;
                    unsigned long const g_col = std::accumulate( grad.shape().begin()+ax, grad.shape().end(), 1UL, []( unsigned long x, unsigned long y ){ return x*y; } );
                    unsigned long const g_row = grad.size() / g_col;
                    view_2d<value_type> v_g{ grad.data(), g_row, g_col };

                    // 2D view of l_ans
                    unsigned long const lhs_row = g_row;
                    unsigned long const lhs_col = lhs_input.size() / lhs_row;
                    view_2d<value_type> v_l{ l_ans.data(), lhs_row, lhs_col };

                    // 2D view of r_ans
                    unsigned long const rhs_row = g_row;
                    unsigned long const rhs_col = rhs_input.size() / rhs_row;
                    view_2d<value_type> v_r{ r_ans.data(), rhs_row, rhs_col };
```

```
                    better_assert( g_col == lhs_col + rhs_col, "last dimension not agree" );

                    for ( unsigned long idx = 0; idx != g_row; ++idx )
                    {
                        std::copy( v_g[idx], v_g[idx]+lhs_col, v_l[idx] );          // fill idx-th row of 'v_l'
                        std::copy( v_g[idx]+lhs_col, v_g[idx]+g_col, v_r[idx] );    // fill idx-th row of 'v_r'
                    }

                    return std::make_tuple( l_ans, r_ans );
                },
                "Concatenate",
                [axe]( std::vector<unsigned long> const& l, std::vector<unsigned long> const& r ) noexcept
                {
                    better_assert( l.size() == r.size(), fmt::format( "expecting of same size, but lhs.size is {} and rhs.size is {}.", l.size(), r.size() ) );
                    // more assertion ?
                    std::vector<unsigned long> ans = l;
                    if ( axe > ans.size() ) axe = ans.size() - 1;
                    ans[axe] += r[axe];
                    return ans;
                }
            )( lhs_ex, rhs_ex );
        };
    }

    // just to keep this interface agrees with Keras
    inline auto concatenate( unsigned long axe = -1 )
    {

        return [=]< Expression Lhs_Expression, Expression Rhs_Expression >( Lhs_Expression const& lhs_ex, Rhs_Expression const& rhs_ex ) noexcept
        {
            return concatenate( lhs_ex, rhs_ex )( axe );
        };
    }

    // alias of 'concatenate'
    template< Expression Lhs_Expression, Expression Rhs_Expression >
    auto constexpr concat( Lhs_Expression const& lhs_ex, Rhs_Expression const& rhs_ex ) noexcept
    {
        return concatenate( lhs_ex, rhs_ex )();
    }

    // alias of 'concatenate'
    inline auto concat( unsigned long axe = -1 )
    {
        return concatenate( axe );
    }

    template< Expression Lhs_Expression, Expression Rhs_Expression >
    auto constexpr maximum( Lhs_Expression const& lhs_ex, Rhs_Expression const& rhs_ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> mask_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache_lhs = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache_rhs = std::make_shared<std::any>();
        return make_binary_operator
        (
            [=]<Tensor Tsor>( Tsor const& lhs_tensor, Tsor const& rhs_tensor ) noexcept
            {
                better_assert( lhs_tensor.shape() == rhs_tensor.shape(), "tensor shape mismatch." );

                Tsor& ans = context_cast<Tsor>( forward_cache );
                ans.resize( lhs_tensor.shape() );
                Tsor& mask = context_cast<Tsor>( mask_cache ); // 1 if lhs element is larger, 0 if rhs element is larger
                mask.resize( lhs_tensor.shape() );

                for_each( lhs_tensor.begin(), lhs_tensor.end(), rhs_tensor.begin(), ans.begin(), mask.begin(), []( auto const l, auto const r, auto& a, auto& m ) { m = l > r ?  1.0 :  0.0; a = l > r ?  l :  r; } );

                return ans;
            },
            [=]<Tensor Tsor>( Tsor const& lhs_input, Tsor const& rhs_input, Tsor const&, Tsor const& grad ) noexcept
            {
                Tsor& mask = context_cast<Tsor>( mask_cache ); // 1 if lhs element is larger, 0 if rhs element is larger

                Tsor& l_ans = context_cast<Tsor>( backward_cache_lhs );
                l_ans.resize( lhs_input.shape() );
                Tsor& r_ans = context_cast<Tsor>( backward_cache_rhs );
```

```
                r_ans.resize( rhs_input.shape() );

                for_each( grad.begin(), grad.end(), mask.begin(), l_ans.begin(), r_ans.begin(), []( auto const g, auto const m, auto& l, auto& r ) { if ( m > 0.5 ) { l = g; r = 0.0; } else { l = 0.0; r = g; } } );

                return std::make_tuple( l_ans, r_ans );
            },
            "Maximum"
        )( lhs_ex, rhs_ex );
    }

    template< Expression Lhs_Expression, Expression Rhs_Expression >
    auto constexpr minimum( Lhs_Expression const& lhs_ex, Rhs_Expression const& rhs_ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> mask_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache_lhs = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache_rhs = std::make_shared<std::any>();
        return make_binary_operator
        (
            [=]<Tensor Tsor>( Tsor const& lhs_tensor, Tsor const& rhs_tensor ) noexcept
            {
                better_assert( lhs_tensor.shape() == rhs_tensor.shape(), "tensor shape mismatch." );

                Tsor& ans = context_cast<Tsor>( forward_cache );
                ans.resize( lhs_tensor.shape() );
                Tsor& mask = context_cast<Tsor>( mask_cache ); // 1 if lhs element is larger, 0 if rhs element is larger
                mask.resize( lhs_tensor.shape() );

                for_each( lhs_tensor.begin(), lhs_tensor.end(), rhs_tensor.begin(), ans.begin(), mask.begin(), []( auto const l, auto const r, auto& a, auto& m ) { m = l > r ?  0.0:  1.0 ; a = l > r ?  r:  l; } );

                return ans;
            },
            [=]<Tensor Tsor>( Tsor const& lhs_input, Tsor const& rhs_input, Tsor const&, Tsor const& grad ) noexcept
            {
                Tsor& mask = context_cast<Tsor>( mask_cache ); // 1 if lhs element is larger, 0 if rhs element is larger

                Tsor& l_ans = context_cast<Tsor>( backward_cache_lhs );
                l_ans.resize( lhs_input.shape() );
                Tsor& r_ans = context_cast<Tsor>( backward_cache_rhs );
                r_ans.resize( rhs_input.shape() );

                for_each( grad.begin(), grad.end(), mask.begin(), l_ans.begin(), r_ans.begin(), []( auto const g, auto const m, auto& l, auto& r ) { if ( m < 0.5 ) { l = g; r = 0.0; } else { l = 0.0; r = g; } } );

                return std::make_tuple( l_ans, r_ans );
            },
            "Minmum"
        )( lhs_ex, rhs_ex );
    }

    template< Expression Lhs_Expression, Expression Rhs_Expression >
    auto constexpr atan2( Lhs_Expression const& lhs_ex, Rhs_Expression const& rhs_ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache_lhs = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache_rhs = std::make_shared<std::any>();
        return make_binary_operator
        (
            [=]<Tensor Tsor>( Tsor const& lhs_tensor, Tsor const& rhs_tensor ) noexcept
            {
                better_assert( lhs_tensor.shape() == rhs_tensor.shape(), "tensor shape mismatch." );
                Tsor& ans = context_cast<Tsor>( forward_cache );
                ans.resize( lhs_tensor.shape() );
                for_each( lhs_tensor.begin(), lhs_tensor.end(), rhs_tensor.begin(), ans.begin(), []( auto const l, auto const r, auto& a ) { a = std::atan2(l, r); } );
                return ans;
            },
            [=]<Tensor Tsor>( Tsor const& lhs_input, Tsor const& rhs_input, Tsor const&, Tsor const& grad ) noexcept
            {
                Tsor& l_ans = context_cast<Tsor>( backward_cache_lhs );
                l_ans.resize( lhs_input.shape() );
                Tsor& r_ans = context_cast<Tsor>( backward_cache_rhs );
                r_ans.resize( rhs_input.shape() );
                for_each( grad.begin(), grad.end(), l_ans.begin(), r_ans.begin(), lhs_input.begin(), rhs_input.begin(), []( auto const g, auto& l, auto& r, auto const x, auto const y ) { auto const c = x*x+y*y; l = -g*y/c; r = g*x/c; } );
                return std::make_tuple( l_ans, r_ans );
            },
```

```
            "Arctan2"
        )( lhs_ex, rhs_ex );
    }


    template< typename T=float > requires std::floating_point<T>
    inline auto random_normal_like( T mean = 0.0, T stddev = 1.0 ) noexcept
    {
        return [=]<Expression Ex>(Ex const& ex ) noexcept
        {
            return make_unary_operator
            (
                [=]<Tensor Tsor>( Tsor const& tsor ) noexcept
                {
                    return randn_like( tsor, mean, stddev );
                },
                []<Tensor Tsor>( Tsor const&, Tsor const&, Tsor const& grad ) noexcept
                {
                    return zeros_like( grad );
                },
                "RandomNormalLike"
            )(ex);
        };
    }

    template< Expression Ex>
    auto ones_like( Ex const& ex ) noexcept
    {
        return make_unary_operator
        (
            []<Tensor Tsor>( Tsor const& tsor ) noexcept { return ones_like( tsor ); },
            []<Tensor Tsor>( Tsor const&, Tsor const& , Tsor const& grad ) noexcept { return zeros_like( grad ); },
            "OnesLike"
        )(ex);
    }

    template< Expression Ex>
    auto zeros_like( Ex const& ex ) noexcept
    {
        return make_unary_operator
        (
            []<Tensor Tsor>( Tsor const& tsor ) noexcept { return zeros_like( tsor ); },
            []<Tensor Tsor>( Tsor const&, Tsor const& , Tsor const& grad ) noexcept { return zeros_like( grad ); },
            "ZerosLike"
        )(ex);
    }

    template< Expression Lhs_Expression, Expression Rhs_Expression, std::floating_point FP >
    auto constexpr equal( Lhs_Expression const& lhs_ex, Rhs_Expression const& rhs_ex, FP threshold=0.5 ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();
        return make_binary_operator
        (
            [=]<Tensor Tsor>( Tsor const& lhs_tensor, Tsor const& rhs_tensor ) noexcept
            {
                typedef typename Tsor::value_type value_type;
                better_assert( lhs_tensor.shape() == rhs_tensor.shape(), "equal:  tensor shape mismatch." );

                Tsor& ans = context_cast<Tsor>( forward_cache );
                ans.resize( lhs_tensor.shape() );
                for_each( lhs_tensor.begin(), lhs_tensor.end(), rhs_tensor.begin(), ans.begin(), [threshold]( auto l, auto r, auto& v ){ v = (std::abs(l-r) > threshold) ?  value_type{0} : value_type{1}; } );
                return ans;
            },
            [=]<Tensor Tsor>( Tsor const& lhs_input, Tsor const& rhs_input, Tsor const&, Tsor const& grad ) noexcept
            {
                typedef typename Tsor::value_type value_type;
                Tsor& ans = context_cast<Tsor>( backward_cache );
                std::fill( ans.begin(), ans.end(), value_type{0} );
                return std::make_tuple( ans, ans );
            },
            "Equal"
        )( lhs_ex, rhs_ex );
    }

    template <Expression Ex>
    auto constexpr sign( Ex const& ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();
```

```
        return make_unary_operator
        (
            [=]<Tensor Tsor>( Tsor const& input ) noexcept
            {
                typedef typename Tsor::value_type value_type;
                Tsor& ans = context_cast<Tsor>( forward_cache );
                ans.resize( input.shape() );
                for_each( input.begin(), input.end(), ans.begin(), []( auto x, auto& v ){ v = (value_type{0} < x) - (x < value_type{0}); } );
                return ans;
            },
            [=]<Tensor Tsor>( Tsor const&input, Tsor const&, Tsor const& grad ) noexcept
            {
                typedef typename Tsor::value_type value_type;
                Tsor& ans = context_cast<Tsor>( backward_cache );
                ans.resize( input.shape() );
                std::fill( ans.begin(), ans.end(), value_type{0} ); //TF gives zeros, we follow TF here
                return ans;
            },
            "Sign"
        )( ex );
    };



    namespace
    {
        struct zero_padding_2d_context
        {
            auto make_forward() const noexcept
            {
                return []( unsigned long top, unsigned long bottom, unsigned long left, unsigned long right, std::shared_ptr<std::any> forward_cache ) noexcept
                {
                    return [=]<Tensor Tsor>( Tsor const& input ) noexcept
                    {
                        typedef typename Tsor::value_type value_type;
                        better_assert( input.ndim() == 4, "Expecting a 4D tensor, but got ", input.ndim() );

                        // 4D view of input tensor
                        std::vector<unsigned long> shape = input.shape();
                        auto const[batch_size, row, col, channel] = std::make_tuple(shape[0], shape[1], shape[2], shape[3]);
                        Tsor input_ = input;
                        view_4d<value_type> ts{ input_.data(), batch_size, row, col, channel };

                        // 4D view of output tensor
                        Tsor& ans = context_cast<Tsor>( forward_cache );
                        ans.resize( {batch_size, top+row+bottom, left+col+right, channel} );
                        view_4d<value_type> ta{ ans.data(), batch_size, top+row+bottom, left+col+right, channel };

                        for ( auto bs :  range( batch_size ) )
                            for ( auto r :  range( row ) )
                                for ( auto c :  range( col ) )
                                    for ( auto ch :  range( channel ) )
                                        ta[bs][top+r][left+c][ch] = ts[bs][r][c][ch];

                        return ans;
                    };
                };
            }

            auto make_backward() const noexcept
            {
                return []( unsigned long top, unsigned long bottom, unsigned long left, unsigned long right, std::shared_ptr<std::any> backward_cache ) noexcept
                {
                    return [=]<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor const& grad ) noexcept
                    {
                        typedef typename Tsor::value_type value_type;
                        std::vector<unsigned long> const& shape = input.shape();
                        auto const[batch_size, row, col, channel] = std::make_tuple(shape[0], shape[1], shape[2], shape[3]);

                        Tsor& ans = context_cast<Tsor>( backward_cache );
                        ans.resize( input.shape() );
                        std::fill( ans.begin(), ans.end(), value_type{0} );

                        view_4d<value_type> ta{ ans.data(), batch_size, row, col, channel };

                        Tsor grad_ = grad;
                        view_4d<value_type> tg{ grad_.data(), batch_size, top+row+bottom, left+col+right, channel };

```

```
                        for ( auto bs : range( batch_size ) )
                            for ( auto r : range( row ) )
                                for ( auto c : range( col ) )
                                    for ( auto ch : range( channel ) )
                                        ta[bs][r][c][ch] = tg[bs][r+top][c+left][ch];
                        return ans;
                    };
                };
            }
        }; // zero_padding_2d_context
    }//anonymouse namespace

    inline auto zero_padding_2d( std::vector<unsigned long> const& padding ) noexcept
    {
        // extracting paddings
        unsigned long top, bottom, left, right;
        if ( padding.size() == 1 )
            std::tie( top, bottom, left, right ) = std::make_tuple( padding[0], padding[0], padding[0], padding[0] );
        else if (padding.size() == 2 )
            std::tie( top, bottom, left, right ) = std::make_tuple( padding[0], padding[0], padding[1], padding[1] );
        else if (padding.size() == 4 )
            std::tie( top, bottom, left, right ) = std::make_tuple( padding[0], padding[1], padding[2], padding[3] );
        else
            better_assert( false, "Expecting padding has size of 1, 2 or 4, but got:  ", padding.size() );

        // checking extracted paddings
        better_assert( top >= 1, "Expecting zero_padding_2d top padding no less than 1, but got ", top );
        better_assert( bottom >= 1, "Expecting zero_padding_2d bottom padding no less than 1, but got ", bottom );
        better_assert( left >= 1, "Expecting zero_padding_2d left padding no less than 1, but got ", left );
        better_assert( right >= 1, "Expecting zero_padding_2d right padding no less than 1, but got ", right );

        // to avoid re-allocating memory for tensors
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();

        return [top, bottom, left, right, forward_cache, backward_cache]<Expression Ex>( Ex const& ex ) noexcept
        {
            return make_unary_operator
            (
                zero_padding_2d_context{}.make_forward()( top, bottom, left, right, forward_cache ),
                zero_padding_2d_context{}.make_backward()( top, bottom, left, right, backward_cache ),
                "ZeroPadding2D",
                [=]( std::vector<unsigned long> const& shape ) noexcept { return std::vector<unsigned long>{ {shape[0], shape[1]+top+bottom, shape[2]+left+right, shape[3]} }; }
            )( ex );
        };
    }



    namespace
    {
        struct cropping_2d_context
        {
            auto make_forward() const noexcept
            {
                return []( unsigned long top, unsigned long bottom, unsigned long left, unsigned long right, std::shared_ptr<std::any> forward_cache ) noexcept
                {
                    return [=]<Tensor Tsor>( Tsor const& input ) noexcept
                    {
                        typedef typename Tsor::value_type value_type;
                        better_assert( input.ndim() == 4, "Expecting a 4D tensor, but got ", input.ndim() );
                        // check shape, not too large

                        // 4D view of input tensor
                        std::vector<unsigned long> shape = input.shape();
                        auto const[batch_size, row, col, channel] = std::make_tuple(shape[0], shape[1], shape[2], shape[3]);
                        Tsor input_ = input;
                        view_4d<value_type> ts{ input_.data(), batch_size, row, col, channel };

                        better_assert( row-top-bottom > 0, fmt::format("Cropping2D: expecting a smaller cropping dimension in row:  row:{}, top:{}, bottop:{}", row, top, bottom ) );
                        better_assert( col-left-right > 0, fmt::format("Cropping2D: expecting a smaller cropping dimension in col:  col:{}, left:{}, right:{}", col, left, right ) );

```

```
                    // 4D view of output tensor
                    Tsor& ans = context_cast<Tsor>( forward_cache );
                    ans.resize( {batch_size, row-top-bottom, col-left-right, channel} );
                    view_4d<value_type> ta{ ans.data(), batch_size, row-top-bottom, col-left-right, channel };

                    for ( auto bs :  range( batch_size ) )
                        for ( auto r :  range( row-top-bottom ) )
                            for ( auto c :  range( col-left-right ) )
                                for ( auto ch :  range( channel ) )
                                    ta[bs][r][c][ch] = ts[bs][top+r][left+c][ch];

                    return ans;
                };
            };
        }

        auto make_backward() const noexcept
        {
            return []( unsigned long top, unsigned long bottom, unsigned long left, unsigned long right, std::shared_ptr<std::any> backward_cache ) noexcept
            {
                return [=]<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor const& grad ) noexcept
                {
                    typedef typename Tsor::value_type value_type;
                    std::vector<unsigned long> const& shape = grad.shape();
                    auto const[batch_size, row, col, channel] = std::make_tuple(shape[0], shape[1], shape[2], shape[3]);

                    Tsor& ans = context_cast<Tsor>( backward_cache );
                    ans.resize( input.shape() );
                    std::fill( ans.begin(), ans.end(), value_type{0} );

                    view_4d<value_type> ta{ ans.data(), batch_size, row+top+bottom, col+left+right, channel };

                    Tsor grad_ = grad;
                    view_4d<value_type> tg{ grad_.data(), batch_size, row, col, channel };

                    for ( auto bs :  range( batch_size ) )
                        for ( auto r :  range( row ) )
                            for ( auto c :  range( col ) )
                                for ( auto ch :  range( channel ) )
                                    ta[bs][r+top][c+left][ch] = tg[bs][r][c][ch];
                    return ans;
                };
            };
        }
    }; // cropping_2d_context
}//anonymouse namespace

inline auto cropping_2d( std::vector<unsigned long> const& padding ) noexcept
{
    // extracting paddings
    unsigned long top, bottom, left, right;
    if ( padding.size() == 1 )
        std::tie( top, bottom, left, right ) = std::make_tuple( padding[0], padding[0], padding[0], padding[0] );
    else if (padding.size() == 2 )
        std::tie( top, bottom, left, right ) = std::make_tuple( padding[0], padding[0], padding[1], padding[1] );
    else if (padding.size() == 4 )
        std::tie( top, bottom, left, right ) = std::make_tuple( padding[0], padding[1], padding[2], padding[3] );
    else
        better_assert( false, "Expecting padding has size of 1, 2 or 4, but got:  ", padding.size() );

    // checking extracted paddings
    better_assert( top >= 1, "Expecting cropping_2d top padding no less than 1, but got ", top );
    better_assert( bottom >= 1, "Expecting cropping_2d bottom padding no less than 1, but got ", bottom );
    better_assert( left >= 1, "Expecting cropping_2d left padding no less than 1, but got ", left );
    better_assert( right >= 1, "Expecting cropping_2d right padding no less than 1, but got ", right );

    // to avoid re-allocating memory for tensors
    std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
    std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();

    return [top, bottom, left, right, forward_cache, backward_cache]<Expression Ex>( Ex const& ex ) noexcept
    {
        return make_unary_operator
        (
```

```
                cropping_2d_context{}.make_forward()( top, bottom, left, right, forward_cache ),
                cropping_2d_context{}.make_backward()( top, bottom, left, right, backward_cache ),
                "ZeroPadding2D",
                [=]( std::vector<unsigned long> const& shape ) noexcept
                {
                    return std::vector<unsigned long>{ {shape[0], shape[1]-top-bottom, shape[2]-left-right, shape[3]} };
                }
            )( ex );
        };
    }


    namespace
    {

        inline auto detailed_sliding_2d( unsigned long const pixels, std::shared_ptr<std::any> shift_cache,
                                        std::shared_ptr<std::any> forward_cache, std::shared_ptr<std::any> backward_cache) noexcept
        {
            return [=]<Expression Ex>( Ex const& ex ) noexcept // <- the output has been zero-padded by n pixels
            {
                return make_unary_operator
                (
                    [=]<Tensor Tsor>( Tsor const& tsor ) noexcept
                    {
                        if (learning_phase != 1)
                            return tsor;

                        typedef typename Tsor::value_type value_type;
                        std::vector<unsigned long> const& shape = tsor.shape();
                        auto const[batch_size, row, col, channel] = std::make_tuple(shape[0], shape[1], shape[2], shape[3]);
                        view_4d vi{tsor.data(), batch_size, row, col, channel};

                        tensor<long> shifts = context_cast<tensor<long»( shift_cache );
                        shifts.resize( {channel, 2} );
                        {   //generating random shifts
                            std::uniform_int_distribution<long> distribution( -pixels, pixels );
                            for ( auto& v :  shifts )
                                v = distribution(random_generator);
                        }
                        view_2d _shifts{shifts.data(), channel, 2};

                        Tsor& ans = context_cast<Tsor>( forward_cache );
                        ans.resize( tsor.shape() );
                        std::fill( ans.begin(), ans.end(), value_type{0} );
                        view_4d vo{ans.data(), batch_size, row, col, channel};

                        for ( auto bs :  range(batch_size ) )
                        {
                            for ( auto ch :  range( channel ) )
                            {
                                auto [row_shift, col_shift] = std::make_tuple( _shifts[ch][0], _shifts[ch][1]);
                                for ( auto r :  range( row ) )
                                {
                                    if (r-row_shift>=0 && r-row_shift<row)
                                    {
                                        for ( auto c :  range( col ) )
                                        {
                                            if (c-col_shift>=0 && c-col_shift<col )
                                                vo[bs][r][c][ch] = vi[bs][r-row_shift][c-col_shift][ch];
                                        }
                                    }
                                }
                            }
                        }
                        return ans;
                    },
                    [=]<Tensor Tsor>( Tsor const&, Tsor const&, Tsor const& grad ) noexcept
                    {
                        typedef typename Tsor::value_type value_type;
                        std::vector<unsigned long> const& shape = grad.shape();
                        auto const[batch_size, row, col, channel] = std::make_tuple( shape[0], shape[1], shape[2], shape[3] );
                        view_4d vi{grad.data(), batch_size, row, col, channel};
                        tensor<long> shifts = context_cast<tensor<long»( shift_cache );
                        view_2d _shifts{shifts.data(), channel, 2};

                        Tsor& ans = context_cast<Tsor>( backward_cache );
                        ans.resize( grad.shape() );
                        std::fill( ans.begin(), ans.end(), value_type{0} );
                        view_4d vo{ans.data(), batch_size, row, col, channel};
```

```
                        for ( auto bs :  range(batch_size ) )
                        {
                            for ( auto ch :  range( channel ) )
                            {
                                auto [row_shift, col_shift] = std::make_tuple( _shifts[ch][0], _shifts[ch][1] );
                                for ( auto r :  range( row ) )
                                {
                                    if (r+row_shift>=0 && r+row_shift<row)
                                    {
                                        for ( auto c :  range( col ) )
                                        {
                                            if (c+col_shift>=0 && c+col_shift<col )
                                                vo[bs][r][c][ch] = vi[bs][r+row_shift][c+col_shift][ch];
                                        }
                                    }
                                }
                            }
                        }
                        return ans;
                    },
                    "Sliding2D"
                )( ex );
            };
        }

    } // anonymous namespace

    inline auto sliding_2d( unsigned long pixels ) noexcept
    {
        std::shared_ptr<std::any> shift_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();

        return [=]<Expression Ex>( Ex const& ex ) noexcept
        {
            return cropping_2d( {pixels,} )( detailed_sliding_2d(pixels, shift_cache, forward_cache, backward_cache)( zero_padding_2d( {pixels,} )( ex ) )  );
        };
    }

    namespace
    {
        struct repeat_context
        {
            auto make_forward() const noexcept
            {
                return []( unsigned long repeats, unsigned long axis, std::shared_ptr<std::any> forward_cache ) noexcept
                {
                    return [=]<Tensor Tsor>( Tsor const& input ) noexcept
                    {
                        if ( 1UL == repeats ) return input;
                        unsigned long const ax = std::min( axis, input.shape().size()-1 );

                        auto const& shape = input.shape();
                        unsigned long const stride = std::accumulate( shape.begin()+ax+1, shape.end(), 1UL, []( unsigned long x, unsigned long y ){ return x*y; } );
                        unsigned long const iterations = std::accumulate( shape.begin(), shape.begin()+ax+1, 1UL, []( unsigned long x, unsigned long y ){ return x*y; } );

                        // generate output tensor
                        std::vector<unsigned long> output_shape = input.shape();
                        output_shape[ax] *= repeats;

                        Tsor& ans = context_cast<Tsor>( forward_cache );
                        ans.resize( output_shape );

                        // create 2D and 3D view
                        view_2d v2{ input.data(), iterations, stride };
                        view_3d v3{ ans.data(), iterations, repeats, stride };

                        // copy data
                        for ( auto it :  range( iterations ) )
                            for ( auto re :  range( repeats ) )
                                std::copy_n( v2[it], stride, v3[it][re] );

                        return ans;
                    };
                };
            }

            auto make_backward() const noexcept
            {
                return []( unsigned long repeats, unsigned long axis, std::shared_ptr<std::any>
```

```
backward_cache ) noexcept
                {
                    return [=]<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor const& grad ) noexcept
                    {
                        if ( 1UL == repeats ) return grad;
                        unsigned long const ax = std::min( axis, input.shape().size()-1 );

                        auto const& shape = input.shape();
                        unsigned long const stride = std::accumulate( shape.begin()+ax+1, shape.end(), 1UL, []( unsigned long x, unsigned long y ){ return x*y; } );
                        unsigned long const iterations = std::accumulate( shape.begin(), shape.begin()+ax+1, 1UL, []( unsigned long x, unsigned long y ){ return x*y; } );

                        Tsor& ans = context_cast<Tsor>( backward_cache );
                        ans.resize( input.shape() );
                        ans.reset();

                        view_2d v2{ans.data(), iterations, stride };
                        view_3d v3{ grad.data(), iterations, repeats, stride };

                        for ( auto id :  range( iterations ) )
                            for ( auto re :  range( repeats ) )
                                for ( auto st :  range( stride ) )
                                    v2[id][st] += v3[id][re][st];

                        return ans;
                    };
                };
            }
        };//struct repeat_context
    }//anonymous namespace


    inline auto repeat( unsigned long repeats, unsigned long axis=-1 ) noexcept
    {
        better_assert( repeats > 0, "repeat:  repeats can not be zero." );

        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();

        return [repeats, axis, forward_cache, backward_cache]<Expression Ex>( Ex const& ex ) noexcept
        {
            return make_unary_operator
            (
                repeat_context{}.make_forward()( repeats, axis, forward_cache ),
                repeat_context{}.make_backward()( repeats, axis, backward_cache ),
                "Repeat",
                [=]( std::vector<unsigned long> const& shape ) noexcept
                {
                    std::vector<unsigned long> ans = shape;
                    if ( axis >= ans.size() ) axis = ans.size()-1;
                    ans[axis] *= repeats;
                    return ans;
                }
            )
            ( ex );
        };
    }


    namespace
    {
        struct reduce_min_context
        {
            auto make_forward() const noexcept
            {
                return []( unsigned long axis, std::shared_ptr<std::any> forward_cache, std::shared_ptr<std::any> index_cache ) noexcept
                {
                    return [=]<Tensor Tsor>( Tsor const& input ) noexcept
                    {
                        unsigned long const ax = std::min( axis, input.shape().size()-1 );

                        // example:  for an input tensor of shape ( 2, 3, 4, 5 ), and axis is 1
                        auto const& shape = input.shape(); // example:  the shape is ( 2, 3, 4, 5 )
                        unsigned long const stride = std::accumulate( shape.begin()+ax+1, shape.end(), 1UL, []( unsigned long x, unsigned long y ){ return x*y; } ); // example:  the stride is 20
                        unsigned long const iterations = std::accumulate( shape.begin(), shape.begin()+ax, 1UL, []( unsigned long x, unsigned long y ){ return x*y; } ); // example:  the iterations is 2
                        unsigned long const scales = shape[ax]; // the elements in the dimenstion to reduce.  example:  scales is 3

                        // generate output tensor
                        std::vector<unsigned long> output_shape = input.shape(); // example: temporately being ( 2, 3, 4, 5 )
```

```
                        std::copy( output_shape.begin()+ax+1, output_shape.end(), output_shape.begin()+ax ); // example:  temporately being ( 2, 4, 5, 5 )
                        output_shape.resize( output_shape.size() - 1 ); // example:  output_shape is ( 2, 4, 5 )

                        Tsor& ans = context_cast<Tsor>( forward_cache );
                        ans.resize( output_shape ); // example:  ans shape is ( 2, 4, 5 )

                        tensor<unsigned long>& index = context_cast<tensor<unsigned long>>( index_cache );
                        index.resize( output_shape ); // example:  index shape is ( 2, 4, 5 )

                        // create 2D and 3D view
                        view_2d v2{ ans.data(), iterations, stride }; // example:  viewing as a matrix of shape ( 2, 20 )
                        view_2d v_index{ index.data(), iterations, stride }; // example:  viewing as a matrix of ( 2, 20 )
                        view_3d v3{ input.data(), iterations, scales, stride }; // example:  viewing as a tube of ( 2, 3, 20 )

                        // reduce minimal elements along the selected axis
                        for ( auto it :  range( iterations ) ) // example:  range (2)
                            for ( auto st :  range( stride ) ) // example:  range (20)
                            {
                                // reduce the minimal elements along the column of st
                                auto min_itor = std::min_element( v3[it].col_begin(st), v3[it].col_end(st) );
                                v2[it][st] = *min_itor;

                                // record the minimal position offset with respect to the head of the column
                                unsigned long const offset = std::distance( v3[it].col_begin(st), min_itor );
                                v_index[it][st] = offset;
                            }

                        return ans;
                    };
                };
            }

            auto make_backward() const noexcept
            {
                return []( unsigned long axis, std::shared_ptr<std::any> backward_cache, std::shared_ptr<std::any> index_cache ) noexcept
                {
                    return [=]<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor const& grad ) noexcept
                    {
                        unsigned long const ax = std::min( axis, input.shape().size()-1 );

                        // example:  for an input tensor of shape ( 2, 3, 4, 5 ), and axis is 1
                        auto const& shape = input.shape(); // example:  the shape is ( 2, 3, 4, 5 )
                        unsigned long const stride = std::accumulate( shape.begin()+ax+1, shape.end(), 1UL, []( unsigned long x, unsigned long y ){ return x*y; } ); // example:  the stride is 20
                        unsigned long const iterations = std::accumulate( shape.begin(), shape.begin()+ax, 1UL, []( unsigned long x, unsigned long y ){ return x*y; } ); // example:  the iterations is 2
                        unsigned long const scales = shape[ax]; // the elements in the dimension to reduce.  example:  scales is 3

                        std::vector<unsigned long> const& output_shape = grad.shape(); // example:  output shape of ( 2, 4, 5 )
                        tensor<unsigned long>& index = context_cast<tensor<unsigned long>>( index_cache );
                        index.resize( output_shape ); // example:  index shape is ( 2, 4, 5 )

                        Tsor& ans = context_cast<Tsor>( backward_cache );
                        ans.resize( shape ); // example:  ans shape is ( 2, 3, 4, 5 )
                        ans.reset();

                        view_2d v_index{ index.data(), iterations, stride }; // example:  viewing as a matrix of ( 2, 20 )
                        view_3d v3{ ans.data(), iterations, scales, stride }; // example:  view as a cube of ( 2, 3, 20 )
                        view_2d v2{ grad.data(), iterations, stride }; // example:  viewing as a matrix of ( 2, 20 )

                        for ( auto it :  range( iterations ) ) // example:  range( 2 )
                            for ( auto st :  range( stride ) ) // example:  range( 20 )
                            {
                                unsigned long const offset = v_index[it][st]; // get the offset from record
                                v3[it][offset][st] = v2[it][st]; // only the element at the minimal position has gradient back-propagated
                            }

```

```
                        return ans;
                    };
                };
            }
        };//struct reduce_min_context
    }//anonymous namespace


    inline auto reduce_min( unsigned long axis=-1 ) noexcept
    {
        std::shared_ptr<std::any> index_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();

        return [axis, index_cache, forward_cache, backward_cache]<Expression Ex>( Ex const& ex ) noexcept
        {
            return make_unary_operator
            (
                reduce_min_context{}.make_forward()( axis, forward_cache, index_cache ),
                reduce_min_context{}.make_backward()( axis, backward_cache, index_cache ),
                "ReduceMin",
                [=]( std::vector<unsigned long> const& shape ) noexcept
                {
                    std::vector<unsigned long> ans = shape;
                    if ( axis >= shape.size() ) axis = shape.size() - 1;
                    std::copy( ans.begin()+axis+1, ans.end(), ans.begin()+axis );
                    ans.resize( ans.size() - 1 );
                    return ans;
                }
            )
            ( ex );
        };
    }



    namespace
    {
        struct reduce_max_context
        {
            auto make_forward() const noexcept
            {
                return []( unsigned long axis, std::shared_ptr<std::any> forward_cache, std::shared_ptr<std::any> index_cache ) noexcept
                {
                    return [=]<Tensor Tsor>( Tsor const& input ) noexcept
                    {
                        unsigned long const ax = std::min( axis, input.shape().size()-1 );

                        // example:  for an input tensor of shape ( 2, 3, 4, 5 ), and axis is 1
                        auto const& shape = input.shape(); // example:  the shape is ( 2, 3, 4, 5 )
                        unsigned long const stride = std::accumulate( shape.begin()+ax+1, shape.end(), 1UL, []( unsigned long x, unsigned long y ){ return x*y; } ); // example:  the stride is 20
                        unsigned long const iterations = std::accumulate( shape.begin(), shape.begin()+ax, 1UL, []( unsigned long x, unsigned long y ){ return x*y; } ); // example:  the iterations is 2
                        unsigned long const scales = shape[ax]; // the elements in the dimenstion to reduce.  example:  scales is 3

                        // generate output tensor
                        std::vector<unsigned long> output_shape = input.shape(); // example:  temporately being ( 2, 3, 4, 5 )
                        std::copy( output_shape.begin()+ax+1, output_shape.end(), output_shape.begin()+ax ); // example:  temporately being ( 2, 4, 5, 5 )
                        output_shape.resize( output_shape.size() - 1 ); // example:  output_shape is ( 2, 4, 5 )

                        Tsor& ans = context_cast<Tsor>( forward_cache );
                        ans.resize( output_shape ); // example:  ans shape is ( 2, 4, 5 )

                        tensor<unsigned long>& index = context_cast<tensor<unsigned long>>( index_cache );
                        index.resize( output_shape ); // example:  index shape is ( 2, 4, 5 )

                        // create 2D and 3D view
                        view_2d v2{ ans.data(), iterations, stride }; // example:  viewing as a matrix of shape ( 2, 20 )
                        view_2d v_index{ index.data(), iterations, stride }; // example:  viewing as a matrix of ( 2, 20 )
                        view_3d v3{ input.data(), iterations, scales, stride }; // example:  viewing as a tube of ( 2, 3, 20 )

                        // reduce maximal elements along the selected axis
                        for ( auto it :  range( iterations ) ) // example:  range (2)
                            for ( auto st :  range( stride ) ) // example:  range (20)
                            {
```

```
                                // reduce the maximal elements along the column of st
                                auto max_itor = std::max_element( v3[it].col_begin(st), v3[it].col_end(st) );
                                v2[it][st] = *max_itor;

                                // record the maximal position offset with respect to the head of the column
                                unsigned long const offset = std::distance( v3[it].col_begin(st), max_itor );
                                v_index[it][st] = offset;
                            }

                        return ans;
                    };
                };
            }

            auto make_backward() const noexcept
            {
                return []( unsigned long axis, std::shared_ptr<std::any> backward_cache, std::shared_ptr<std::any> index_cache ) noexcept
                {
                    return [=]<Tensor Tsor>( Tsor const& input, Tsor const& , Tsor const& grad ) noexcept
                    {
                        unsigned long const ax = std::min( axis, input.shape().size()-1 );

                        // example:  for an input tensor of shape ( 2, 3, 4, 5 ), and axis is 1
                        auto const& shape = input.shape(); // example:  the shape is ( 2, 3, 4, 5 )
                        unsigned long const stride = std::accumulate( shape.begin()+ax+1, shape.end(), 1UL, []( unsigned long x, unsigned long y ){ return x*y; } ); // example:  the stride is 20
                        unsigned long const iterations = std::accumulate( shape.begin(), shape.begin()+ax, 1UL, []( unsigned long x, unsigned long y ){ return x*y; } ); // example:  the iterations is 2
                        unsigned long const scales = shape[ax]; // the elements in the dimension to reduce.  example:  scales is 3

                        std::vector<unsigned long> const& output_shape = grad.shape(); // example: output shape of ( 2, 4, 5 )
                        tensor<unsigned long>& index = context_cast<tensor<unsigned long»( index_cache );
                        index.resize( output_shape ); // example:  index shape is ( 2, 4, 5 )

                        Tsor& ans = context_cast<Tsor>( backward_cache );
                        ans.resize( shape ); // example:  ans shape is ( 2, 3, 4, 5 )
                        ans.reset();

                        view_2d v_index{ index.data(), iterations, stride }; // example:  viewing as a matrix of ( 2, 20 )
                        view_3d v3{ ans.data(), iterations, scales, stride }; // example:  view as a cube of ( 2, 3, 20 )
                        view_2d v2{ grad.data(), iterations, stride }; // example:  viewing as a matrix of ( 2, 20 )

                        for ( auto it :  range( iterations ) ) // example:  range( 2 )
                            for ( auto st :  range( stride ) ) // example:  range( 20 )
                            {
                                unsigned long const offset = v_index[it][st]; // get the offset from record
                                v3[it][offset][st] = v2[it][st]; // only the element at the maximal position has gradient back-propagated
                            }

                        return ans;
                    };
                };
            }
        };//struct reduce_max_context
    }//anonymous namespace


    inline auto reduce_max( unsigned long axis=-1 ) noexcept
    {
        std::shared_ptr<std::any> index_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();

        return [axis, index_cache, forward_cache, backward_cache]<Expression Ex>( Ex const& ex ) noexcept
        {
            return make_unary_operator
            (
                reduce_max_context{}.make_forward()( axis, forward_cache, index_cache ),
                reduce_max_context{}.make_backward()( axis, backward_cache, index_cache ),
                "ReduceMax",
                [=]( std::vector<unsigned long> const& shape ) noexcept
                {
```

```
                std::vector<unsigned long> ans = shape;
                if ( axis >= shape.size() ) axis = shape.size() - 1;
                std::copy( ans.begin()+axis+1, ans.end(), ans.begin()+axis );
                ans.resize( ans.size() - 1 );
                return ans;
            }
        )
        ( ex );
    };
}



namespace
{
    struct reduce_sum_context
    {
        auto make_forward() const noexcept
        {
            return []( unsigned long axis, std::shared_ptr<std::any> forward_cache ) noexcept
            {
                return [=]<Tensor Tsor>( Tsor const& input ) noexcept
                {
                    typedef typename Tsor::value_type value_type;

                    unsigned long const ax = std::min( axis, input.shape().size()-1 );

                    // example:  for an input tensor of shape ( 2, 3, 4, 5 ), and axis is 1
                    auto const& shape = input.shape(); // example:  the shape is ( 2, 3, 4, 5 )
                    unsigned long const stride = std::accumulate( shape.begin()+ax+1, shape.end(), 1UL, []( unsigned long x, unsigned long y ){ return x*y; } ); // example:  the stride is 20
                    unsigned long const iterations = std::accumulate( shape.begin(), shape.begin()+ax, 1UL, []( unsigned long x, unsigned long y ){ return x*y; } ); // example:  the iterations is 2
                    unsigned long const scales = shape[ax]; // the elements in the dimension to reduce.  example:  scales is 3

                    // generate output tensor
                    std::vector<unsigned long> output_shape = input.shape(); // example: temporately being ( 2, 3, 4, 5 )
                    std::copy( output_shape.begin()+ax+1, output_shape.end(), output_shape.begin()+ax ); // example:  temporately being ( 2, 4, 5, 5 )
                    output_shape.resize( output_shape.size() - 1 ); // example:  output_shape is ( 2, 4, 5 )

                    Tsor& ans = context_cast<Tsor>( forward_cache );
                    ans.resize( output_shape ); // example:  ans shape is ( 2, 4, 5 )

                    // create 2D and 3D view
                    view_2d v2{ ans.data(), iterations, stride }; // example:  viewing as a matrix of shape ( 2, 20 )
                    view_3d v3{ input.data(), iterations, scales, stride }; // example:  viewing as a tube of ( 2, 3, 20 )

                    // reduce sum along the selected axis
                    for ( auto it :  range( iterations ) ) // example:  range (2)
                        for ( auto st :  range( stride ) ) // example:  range (20)
                            v2[it][st] = std::accumulate( v3[it].col_begin(st), v3[it].col_end(st), value_type{0} );

                    return ans;
                };
            };
        }

        auto make_backward() const noexcept
        {
            return []( unsigned long axis, std::shared_ptr<std::any> backward_cache ) noexcept
            {
                return [=]<Tensor Tsor>( Tsor const& input, Tsor const& , Tsor const& grad ) noexcept
                {
                    unsigned long const ax = std::min( axis, input.shape().size()-1 );

                    // example:  for an input tensor of shape ( 2, 3, 4, 5 ), and axis is 1
                    auto const& shape = input.shape(); // example:  the shape is ( 2, 3, 4, 5 )
                    unsigned long const stride = std::accumulate( shape.begin()+ax+1, shape.end(), 1UL, []( unsigned long x, unsigned long y ){ return x*y; } ); // example:  the stride is 20
                    unsigned long const iterations = std::accumulate( shape.begin(), shape.begin()+ax, 1UL, []( unsigned long x, unsigned long y ){ return x*y; } ); // example:  the iterations is 2
                    unsigned long const scales = shape[ax]; // the elements in the dimension to reduce.  example:  scales is 3

                    Tsor& ans = context_cast<Tsor>( backward_cache );
                    ans.resize( shape ); // example:  ans shape is ( 2, 3, 4, 5 )
                    ans.reset();
```

```
                        view_3d v3{ ans.data(), iterations, scales, stride }; // example:  view as a cube of ( 2, 3, 20 )
                        view_2d v2{ grad.data(), iterations, stride }; // example:  viewing as a matrix of ( 2, 20 )

                        for ( auto it :  range( iterations ) ) // example:  range( 2 )
                            for ( auto st :  range( stride ) ) // example:  range( 20 )
                                std::fill( v3[it].col_begin( st ), v3[it].col_end( st ), v2[it][st] );

                        return ans;
                    };
                };
            }
        };//struct reduce_sum_context
    }//anonymous namespace


    inline auto reduce_sum( unsigned long axis ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();

        return [axis, forward_cache, backward_cache]<Expression Ex>( Ex const& ex ) noexcept
        {
            return make_unary_operator
            (
                reduce_sum_context{}.make_forward()( axis, forward_cache ),
                reduce_sum_context{}.make_backward()( axis, backward_cache ),
                "ReduceSum",
                [=]( std::vector<unsigned long> const& shape ) noexcept
                {
                    std::vector<unsigned long> ans = shape;
                    if ( axis >= shape.size() ) axis = shape.size() - 1;
                    std::copy( ans.begin()+axis+1, ans.end(), ans.begin()+axis );
                    ans.resize( ans.size() - 1 );
                    return ans;
                }
            )
            ( ex );
        };
    }




    template <Expression Ex>
    auto constexpr abs( Ex const& ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();
        return make_unary_operator( [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( forward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), ans.begin(), []( auto x, auto& v ) noexcept { v = std::abs(x); } );
                                        return ans;
                                    },
                                    [backward_cache]<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor const& grad ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( backward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), grad.begin(), ans.begin(), []( auto x, auto g, auto& v ) noexcept { v = g * ((x > 0.0) ?  1.0 :  ((x < 0.0) ?  -1.0 :  0.0)); } );
                                        return ans;
                                    },
                                    "Abs"
                )( ex );
    };






    template <Expression Ex>
    auto constexpr acos( Ex const& ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();
        return make_unary_operator( [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( forward_cache );
                                        ans.resize( input.shape() );
```

```
                                        for_each( input.begin(), input.end(), ans.begin(), []( auto x, auto& v ) noexcept { v = std::acos(x); } );
                                        return ans;
                                    },
                                    [backward_cache]<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor const& grad ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( backward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), grad.begin(), ans.begin(), []( auto x, auto g, auto& v ) noexcept { v = - g / std::sqrt(1.0-x*x); } );
                                        return ans;
                                    },
                                    "Acos"
                )( ex );
    };

    template <Expression Ex>
    auto constexpr acosh( Ex const& ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();
        return make_unary_operator( [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( forward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), ans.begin(), []( auto x, auto& v ) noexcept { v = std::acosh(x); } );
                                        return ans;
                                    },
                                    [backward_cache]<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor const& grad ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( backward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), grad.begin(), ans.begin(), []( auto x, auto g, auto& v ) noexcept { v = g / std::sqrt(x*x-1.0); } );
                                        return ans;
                                    },
                                    "Acosh"
                )( ex );
    };

    template <Expression Ex>
    auto constexpr asin( Ex const& ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();
        return make_unary_operator( [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( forward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), ans.begin(), []( auto x, auto& v ) noexcept { v = std::asin(x); } );
                                        return ans;
                                    },
                                    [backward_cache]<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor const& grad ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( backward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), grad.begin(), ans.begin(), []( auto x, auto g, auto& v ) noexcept { v = g / std::sqrt(1.0-x*x); } );
                                        return ans;
                                    },
                                    "Asin"
                )( ex );
    };

    template <Expression Ex>
    auto constexpr asinh( Ex const& ex ) noexcept
    {
```

```
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();
        return make_unary_operator( [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( forward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), ans.begin(), []( auto x, auto& v ) noexcept { v = std::asinh(x); } );
                                        return ans;
                                    },
                                    [backward_cache]<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor const& grad ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( backward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), grad.begin(), ans.begin(), []( auto x, auto g, auto& v ) noexcept { v = g / std::sqrt(1.0+x*x); } );
                                        return ans;
                                    },
                                    "Asinh"
                )( ex );
    };

    template <Expression Ex>
    auto constexpr atan( Ex const& ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();
        return make_unary_operator( [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( forward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), ans.begin(), []( auto x, auto& v ) noexcept { v = std::atan(x); } );
                                        return ans;
                                    },
                                    [backward_cache]<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor const& grad ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( backward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), grad.begin(), ans.begin(), []( auto x, auto g, auto& v ) noexcept { v = g / (1.0+x*x); } );
                                        return ans;
                                    },
                                    "Atan"
                )( ex );
    };

    template <Expression Ex>
    auto constexpr atanh( Ex const& ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();
        return make_unary_operator( [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( forward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), ans.begin(), []( auto x, auto& v ) noexcept { v = std::atanh(x); } );
                                        return ans;
                                    },
                                    [backward_cache]<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor const& grad ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( backward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), grad.begin(), ans.begin(), []( auto x, auto g, auto& v ) noexcept { v = g / (1-x*x); } );
                                        return ans;
                                    },
                                    "Atanh"
                )( ex );
    };
```

```
    template <Expression Ex>
    auto constexpr cbrt( Ex const& ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();
        return make_unary_operator( [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( forward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), ans.begin(), []( auto x, auto& v ) noexcept { v = std::cbrt(x); } );
                                        return ans;
                                    },
                                    [backward_cache]<Tensor Tsor>( Tsor const& input, Tsor const& output, Tsor const& grad ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( backward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), output.begin(), grad.begin(), ans.begin(), []( auto, auto o, auto g, auto& v ) noexcept { v = g / (3.0*o*o); } );
                                        return ans;
                                    },
                                    "Cbert"
                )( ex );
    };






    template <Expression Ex>
    auto constexpr ceil( Ex const& ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();
        return make_unary_operator( [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( forward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), ans.begin(), []( auto x, auto& v ) noexcept { v = std::ceil(x); } );
                                        return ans;
                                    },
                                    []<Tensor Tsor>( Tsor const&, Tsor const&, Tsor const& grad ) noexcept
                                    {
                                        return grad;
                                    },
                                    "Ceil"
                )( ex );
    };






    template <Expression Ex>
    auto constexpr cos( Ex const& ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();
        return make_unary_operator( [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( forward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), ans.begin(), []( auto x, auto& v ) noexcept { v = std::cos(x); } );
                                        return ans;
                                    },
                                    [backward_cache]<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor const& grad ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( backward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), grad.begin(), ans.begin(), []( auto x, auto g, auto& v ) noexcept { v = - g * std::sin(x); } );
                                        return ans;
                                    },
                                    "Cos"
                )( ex );
    };
```

```
    template <Expression Ex>
    auto constexpr cosh( Ex const& ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();
        return make_unary_operator( [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( forward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), ans.begin(), []( auto x, auto& v ) noexcept { v = std::cosh(x); } );
                                        return ans;
                                    },
                                    [backward_cache]<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor const& grad ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( backward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), grad.begin(), ans.begin(), []( auto x, auto g, auto& v ) noexcept { v = g * std::sinh(x); } );
                                        return ans;
                                    },
                                    "Cosh"
                )( ex );
    };

    template <Expression Ex>
    auto constexpr erf( Ex const& ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();
        return make_unary_operator( [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( forward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), ans.begin(), []( auto x, auto& v ) noexcept { v = std::erf(x); } );
                                        return ans;
                                    },
                                    [backward_cache]<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor const& grad ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( backward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), grad.begin(), ans.begin(), []( auto x, auto g, auto& v ) noexcept { v = typename Tsor::value_type{1.12837916709551257389} * g * std::exp(-x*x); } );
                                        return ans;
                                    },
                                    "Erf"
                )( ex );
    };

    template <Expression Ex>
    auto constexpr erfc( Ex const& ex ) noexcept
    {

        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();
        return make_unary_operator( [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( forward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), ans.begin(), []( auto x, auto& v ) noexcept { v = std::erfc(x); } );
                                        return ans;
                                    },
                                    [backward_cache]<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor const& grad ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( backward_cache );
                                        ans.resize( input.shape() );
```

```
                                        for_each( input.begin(), input.end(), grad.begin(), ans.begin(), []( auto x, auto g, auto& v ) noexcept { v = typename Tsor::value_type{-1.1283791670955125738 9} * g * std::exp(-x*x); } );
                                        return ans;
                                    },
                                    "Erfc"
                    )( ex );
    };






    template <Expression Ex>
    auto constexpr exp( Ex const& ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();
        return make_unary_operator( [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( forward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), ans.begin(), []( auto x, auto& v ) noexcept { v = std::exp(x); } );
                                        return ans;
                                    },
                                    [backward_cache]<Tensor Tsor>( Tsor const& input, Tsor const& output, Tsor const& grad ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( backward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), output.begin(), grad.begin(), ans.begin(), []( auto, auto o, auto g, auto& v ) noexcept { v = g * o; } );
                                        return ans;
                                    },
                                    "Exp"
                    )( ex );
    };






    template <Expression Ex>
    auto constexpr exp2( Ex const& ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();
        return make_unary_operator( [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( forward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), ans.begin(), []( auto x, auto& v ) noexcept { v = std::exp2(x); } );
                                        return ans;
                                    },
                                    [backward_cache]<Tensor Tsor>( Tsor const& input, Tsor const& output, Tsor const& grad ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( backward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), output.begin(), grad.begin(), ans.begin(), []( auto, auto o, auto g, auto& v ) noexcept { v = std::log(2.0) * g * o; } );
                                        return ans;
                                    },
                                    "Exp2"
                    )( ex );
    };






    template <Expression Ex>
    auto constexpr expm1( Ex const& ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();
        return make_unary_operator( [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( forward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), ans.begin(), []( auto x,
```

```
auto& v ) noexcept { v = std::expm1(x); } );
                                    return ans;
                                },
                                [backward_cache]<Tensor Tsor>( Tsor const& input, Tsor const& output, Tsor const& grad ) noexcept
                                {
                                    Tsor& ans = context_cast<Tsor>( backward_cache );
                                    ans.resize( input.shape() );
                                    for_each( input.begin(), input.end(), output.begin(), grad.begin(), ans.begin(), []( auto, auto o, auto g, auto& v ) noexcept { v = g * (o+1.0); } );
                                    return ans;
                                },
                                "Expm1"
                ) ( ex );
    };

    template <Expression Ex>
    auto constexpr fabs( Ex const& ex ) noexcept
    {
        return abs( ex );
    };

    template <Expression Ex>
    auto constexpr floor( Ex const& ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        return make_unary_operator( [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                {
                                    Tsor& ans = context_cast<Tsor>( forward_cache );
                                    ans.resize( input.shape() );
                                    for_each( input.begin(), input.end(), ans.begin(), []( auto x, auto& v ) noexcept { v = std::floor(x); } );
                                    return ans;
                                },
                                []<Tensor Tsor>( Tsor const&, Tsor const&, Tsor const& grad ) noexcept
                                {
                                    return grad;
                                },
                                "Floor"
                ) ( ex );
    };

    template <Expression Ex>
    auto constexpr llrint( Ex const& ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        return make_unary_operator( [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                {
                                    Tsor& ans = context_cast<Tsor>( forward_cache );
                                    ans.resize( input.shape() );
                                    for_each( input.begin(), input.end(), ans.begin(), []( auto x, auto& v ) noexcept { v = std::llrint(x); } );
                                    return ans;
                                },
                                []<Tensor Tsor>( Tsor const&, Tsor const&, Tsor const& grad ) noexcept
                                {
                                    return grad;
                                },
                                "Llrint"
                ) ( ex );
    };
```

```
    template <Expression Ex>
    auto constexpr llround( Ex const& ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        return make_unary_operator( [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( forward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), ans.begin(), []( auto x, auto& v ) noexcept { v = std::llround(x); } );
                                        return ans;
                                    },
                                    []<Tensor Tsor>( Tsor const&, Tsor const&, Tsor const& grad ) noexcept
                                    {
                                        return grad;
                                    },
                                    "Llround"
                )( ex );
    };






    template <Expression Ex>
    auto constexpr log( Ex const& ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();
        return make_unary_operator( [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( forward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), ans.begin(), []( auto x, auto& v ) noexcept { v = std::log(x); } );
                                        return ans;
                                    },
                                    [backward_cache]<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor const& grad ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( backward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), grad.begin(), ans.begin(), []( auto x, auto g, auto& v ) noexcept { v = g / x; } );
                                        return ans;
                                    },
                                    "Log"
                )( ex );
    };






    template <Expression Ex>
    auto constexpr log10( Ex const& ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();
        return make_unary_operator( [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( forward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), ans.begin(), []( auto x, auto& v ) noexcept { v = std::log10(x); } );
                                        return ans;
                                    },
                                    [backward_cache]<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor const& grad ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( backward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), grad.begin(), ans.begin(), []( auto x, auto g, auto& v ) noexcept { v = g / (2.30258509299404568402*x); } );
                                        return ans;
                                    },
                                    "Log10"
                )( ex );
    };
```

```
    template <Expression Ex>
    auto constexpr log1p( Ex const& ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();
        return make_unary_operator( [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                {
                                    Tsor& ans = context_cast<Tsor>( forward_cache );
                                    ans.resize( input.shape() );
                                    for_each( input.begin(), input.end(), ans.begin(), []( auto x,
auto& v ) noexcept { v = std::log1p(x); } );
                                    return ans;
                                },
                                [backward_cache]<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor
const& grad ) noexcept
                                {
                                    Tsor& ans = context_cast<Tsor>( backward_cache );
                                    ans.resize( input.shape() );
                                    for_each( input.begin(), input.end(), grad.begin(),
ans.begin(), []( auto x, auto g, auto& v ) noexcept { v = g / x; } );
                                    return ans;
                                },
                                "Log1p"
                )( ex );
    };






    template <Expression Ex>
    auto constexpr log2( Ex const& ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();
        return make_unary_operator( [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                {
                                    Tsor& ans = context_cast<Tsor>( forward_cache );
                                    ans.resize( input.shape() );
                                    for_each( input.begin(), input.end(), ans.begin(), []( auto x,
auto& v ) noexcept { v = std::log2(x); } );
                                    return ans;
                                },
                                [backward_cache]<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor
const& grad ) noexcept
                                {
                                    Tsor& ans = context_cast<Tsor>( backward_cache );
                                    ans.resize( input.shape() );
                                    for_each( input.begin(), input.end(), grad.begin(),
ans.begin(), []( auto x, auto g, auto& v ) noexcept { v = g / (0.6931471805599453094*x); } );
                                    return ans;
                                },
                                "Log2"
                )( ex );
    };



    template <Expression Ex>
    auto constexpr lrint( Ex const& ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();
        return make_unary_operator( [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                {
                                    Tsor& ans = context_cast<Tsor>( forward_cache );
                                    ans.resize( input.shape() );
                                    for_each( input.begin(), input.end(), ans.begin(), []( auto x,
auto& v ) noexcept { v = std::lrint(x); } );
                                    return ans;
                                },
                                [backward_cache]<Tensor Tsor>( Tsor const&, Tsor const&, Tsor
const& grad ) noexcept
                                {
                                    return grad;
                                },
                                "Lrint"
                )( ex );
    };
```

```
    template <Expression Ex>
    auto constexpr lround( Ex const& ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        return make_unary_operator( [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( forward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), ans.begin(), []( auto x, auto& v ) noexcept { v = std::lround(x); } );
                                        return ans;
                                    },
                                    []<Tensor Tsor>( Tsor const&, Tsor const&, Tsor const& grad ) noexcept
                                    {
                                        return grad;
                                    },
                                    "Lround"
                )( ex );
    };






    template <Expression Ex>
    auto constexpr nearbyint( Ex const& ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        return make_unary_operator( [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( forward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), ans.begin(), []( auto x, auto& v ) noexcept { v = std::nearbyint(x); } );
                                        return ans;
                                    },
                                    []<Tensor Tsor>( Tsor const&, Tsor const&, Tsor const& grad ) noexcept
                                    {
                                        return grad;
                                    },
                                    "Nearbyint"
                )( ex );
    };






    template <Expression Ex>
    auto constexpr rint( Ex const& ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        return make_unary_operator( [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( forward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), ans.begin(), []( auto x, auto& v ) noexcept { v = std::rint(x); } );
                                        return ans;
                                    },
                                    []<Tensor Tsor>( Tsor const&, Tsor const&, Tsor const& grad ) noexcept
                                    {
                                        return grad;
                                    },
                                    "Rint"
                )( ex );
    };






    template <Expression Ex>
    auto constexpr round( Ex const& ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        return make_unary_operator( [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( forward_cache );
                                        ans.resize( input.shape() );
```

```
                                        for_each( input.begin(), input.end(), ans.begin(), []( auto x, auto& v ) noexcept { v = std::round(x); } );
                                        return ans;
                                    },
                                    []<Tensor Tsor>( Tsor const&, Tsor const&, Tsor const& grad ) noexcept
                                    {
                                        return grad;
                                    },
                                    "Round"
                )( ex );
    };

    template <Expression Ex>
    auto constexpr sin( Ex const& ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();
        return make_unary_operator( [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( forward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), ans.begin(), []( auto x, auto& v ) noexcept { v = std::sin(x); } );
                                        return ans;
                                    },
                                    [backward_cache]<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor const& grad ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( backward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), grad.begin(), ans.begin(), []( auto x, auto g, auto& v ) noexcept { v = g * std::cos(x); } );
                                        return ans;
                                    },
                                    "Sin"
                )( ex );
    };

    template <Expression Ex>
    auto constexpr sinh( Ex const& ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();
        return make_unary_operator( [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( forward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), ans.begin(), []( auto x, auto& v ) noexcept { v = std::sinh(x); } );
                                        return ans;
                                    },
                                    [backward_cache]<Tensor Tsor>( Tsor const& input, Tsor const&, Tsor const& grad ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( backward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), grad.begin(), ans.begin(), []( auto x, auto g, auto& v ) noexcept { v = g * std::cosh(x); } );
                                        return ans;
                                    },
                                    "Sinh"
                )( ex );
    };

    template <Expression Ex>
    auto constexpr sqrt( Ex const& ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();
        return make_unary_operator( [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                    {
```

```
                                        Tsor& ans = context_cast<Tsor>( forward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), ans.begin(), []( auto x, auto& v ) noexcept { v = std::sqrt(x); } );
                                        return ans;
                                    },
                                    [backward_cache]<Tensor Tsor>( Tsor const& input, Tsor const& output, Tsor const& grad ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( backward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), output.begin(), grad.begin(), ans.begin(), []( auto, auto o, auto g, auto& v ) noexcept { v = g / (o+o); } );
                                        return ans;
                                    },
                                    "Sqrt"
                )( ex );
    };

    template <Expression Ex>
    auto constexpr tan( Ex const& ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();
        return make_unary_operator( [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( forward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), ans.begin(), []( auto x, auto& v ) noexcept { v = std::tan(x); } );
                                        return ans;
                                    },
                                    [backward_cache]<Tensor Tsor>( Tsor const& input, Tsor const& output, Tsor const& grad ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( backward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), output.begin(), grad.begin(), ans.begin(), []( auto x, auto o, auto g, auto& v ) noexcept { v = g * (1.0+o*o); } );
                                        return ans;
                                    },
                                    "Tan"
                )( ex );
    };

    template <Expression Ex>
    auto constexpr tanh( Ex const& ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        std::shared_ptr<std::any> backward_cache = std::make_shared<std::any>();
        return make_unary_operator( [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( forward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), ans.begin(), []( auto x, auto& v ) noexcept { v = std::tanh(x); } );
                                        return ans;
                                    },
                                    [backward_cache]<Tensor Tsor>( Tsor const& input, Tsor const& output, Tsor const& grad ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( backward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), output.begin(), grad.begin(), ans.begin(), []( auto, auto o, auto g, auto& v ) noexcept { v = g * (1.0-o*o); } );
                                        return ans;
                                    },
                                    "Tanh"
                )( ex );
    };
```

```
    template <Expression Ex>
    auto constexpr trunc( Ex const& ex ) noexcept
    {
        std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
        return make_unary_operator( [forward_cache]<Tensor Tsor>( Tsor const& input ) noexcept
                                    {
                                        Tsor& ans = context_cast<Tsor>( forward_cache );
                                        ans.resize( input.shape() );
                                        for_each( input.begin(), input.end(), ans.begin(), []( auto x, auto& v ) noexcept { v = std::trunc(x); } );
                                        return ans;
                                    },
                                    []<Tensor Tsor>( Tsor const&, Tsor const&, Tsor const& grad ) noexcept
                                    {
                                        return grad;
                                    },
                                    "Trunc"
                    )( ex );
    };



    template< Expression Lhs_Expression, Variable Rhs_Expression >
    auto constexpr assign( Lhs_Expression const& lhs_ex, Rhs_Expression const& rhs_ex ) noexcept
    {
        return make_binary_operator( []<Tensor Tsor>( Tsor const& lhs_tensor, Tsor& rhs_tensor ) noexcept
                                     {
                                        rhs_tensor.reshape( lhs_tensor.shape() );
                                        std::copy( lhs_tensor.begin(), lhs_tensor.end(), rhs_tensor.begin() );
                                        return lhs_tensor;
                                     },
                                     []<Tensor Tsor>( Tsor const& lhs_input, Tsor const& rhs_input, Tsor const&, Tsor const& ) noexcept
                                     {
                                        return std::make_tuple( zeros_like( lhs_input ), zeros_like( rhs_input ) );
                                     },
                                     "Assign"
                    )( lhs_ex, rhs_ex );
    };



    template< Expression Ex>
    auto poisson(Ex const& ex ) noexcept
    {
        return make_unary_operator
        (
            [=]<Tensor Tsor>( Tsor const& tsor ) noexcept
            {
                return poisson( tsor );
            },
            []<Tensor Tsor>( Tsor const&, Tsor const&, Tsor const& grad ) noexcept
            {
                return grad;
            },
            "Poisson"
        )(ex);
    }


}//namespace ceras

#endif//IPKVWSJOCMGGVRASCBLPYHFBCHRIVEXYBOMMDAKFAUDFYVYOOOISLRXJNUJKPJEVMLDPRDSNM
```

9.25 /home/feng/workspace/github.repo/ceras/include/optimizer.hpp File Reference

```
#include "./config.hpp"
#include "./operation.hpp"
#include "./place_holder.hpp"
#include "./variable.hpp"
```

```
#include "./session.hpp"
#include "./utils/color.hpp"
#include "./utils/debug.hpp"
#include "./utils/id.hpp"
#include "./utils/enable_shared.hpp"
#include "./utils/fmt.hpp"
```

Classes

- struct ceras::sgd< Loss, T >
- struct ceras::adagrad< Loss, T >
- struct ceras::rmsprop< Loss, T >
- struct ceras::adadelta< Loss, T >
- struct ceras::adam< Loss, T >
- struct ceras::gradient_descent< Loss, T >

Namespaces

- namespace ceras

Typedefs

- template<typename Loss , typename T >
  using ceras::ada_grad = adagrad< Loss, T >
- template<typename Loss , typename T >
  using ceras::rms_prop = rmsprop< Loss, T >
- template<typename Loss , typename T >
  using ceras::ada_delta = adadelta< Loss, T >

Variables

- auto ceras::Adam
- auto ceras::SGD
- auto ceras::Adagrad
- auto ceras::RMSprop
- auto ceras::Adadelta

9.26 optimizer.hpp

Go to the documentation of this file.

```
#ifndef XNRPSJMCYFXBDGNRJAWDNDIYQNGNXMRVLEHGNQWILKMTHGNOVHODLLXCCNIMUUFQSMOIYHDUD
#define XNRPSJMCYFXBDGNRJAWDNDIYQNGNXMRVLEHGNQWILKMTHGNOVHODLLXCCNIMUUFQSMOIYHDUD

#include "./config.hpp"
#include "./operation.hpp"
#include "./place_holder.hpp"
#include "./variable.hpp"
#include "./session.hpp"
#include "./utils/color.hpp"
#include "./utils/debug.hpp"
#include "./utils/id.hpp"
#include "./utils/enable_shared.hpp"
#include "./utils/fmt.hpp"
```

```

namespace ceras
{

    // sgd:
    //      - loss:
    //      - batch_size:
    //      - learning_rate:
    //      - momentum:
    //      - decay:  should be very small, such as 1.0e-8
    //      - nesterov:
    //
    template< typename Loss, typename T >
    struct sgd :  enable_id<sgd<Loss, T>, "sgd optimizer">, enable_shared<sgd<Loss, T»
    {
        typedef tensor< T > tensor_type;

        Loss&         loss_;
        T             learning_rate_;
        T             momentum_;
        T             decay_;
        bool          nesterov_;
        unsigned long iterations_;

        sgd(Loss& loss, std::size_t batch_size, T learning_rate=1.0e-1, T momentum=0.0, T decay=0.0, bool nesterov=false) noexcept :
            loss_{loss}, learning_rate_(learning_rate), momentum_(std::max(T{0}, momentum)), decay_{std::max(T{0}, decay)}, nesterov_{nesterov}, iterations_{0}
        {
            better_assert( batch_size >= 1, "batch_size must be positive, but got:  ", batch_size );
            learning_rate_ /= static_cast<T>( batch_size );
        }

        void forward()
        {
            loss_.backward( ones<T>( {1, } ) );
            learning_rate_ /= ( 1.0 + decay_ * iterations_ );
            auto& ss = get_default_session<tensor_type>();
            for ( auto [id, v] :  ss.variables_ )
            {
                if (v.trainable_)
                {
                    auto& data = v.data();
                    auto& gradient = v.gradient();
                    auto& contexts = v.contexts();
                    if ( contexts.empty() ) // create context
                        contexts.push_back( zeros_like( data ) );
                    auto& moments = contexts[0];
                    for_each( moments.begin(), moments.end(), gradient.begin(), [this]( T& m, T g ) { m *= (*this).momentum_; m -= (*this).learning_rate_ * g;} );
                    if (!nesterov_ ) for_each( moments.begin(), moments.end(), data.begin(), gradient.begin(), [this]( T m, T& v, T g ) { v += (*this).momentum_ * m - (*this).learning_rate_ * g; } );
                    else data += moments;

                    gradient.reset(); // clear variable gradient
                }
            }
            ++iterations_;
        }//sgd::forward
    };//sgd

    template< typename Loss, typename T >
    struct adagrad :  enable_id<adagrad<Loss, T >, "adagrad optimizer">, enable_shared<adagrad<Loss,T»
    {
        typedef tensor< T > tensor_type;

        Loss&         loss_;
        T             learning_rate_;
        T             decay_;
        unsigned long iterations_;

        adagrad(Loss& loss, std::size_t batch_size, T learning_rate=1.0e-1, T decay=0.0) noexcept :
                loss_(loss), learning_rate_(learning_rate), decay_{std::max(T{0}, decay)}, iterations_{0}
        {
            better_assert( batch_size >= 1, "batch_size must be positive, but got:  ", batch_size );
            learning_rate_ /= static_cast<T>( batch_size );
        }

        void forward()
        {
            loss_.backward( ones<T>( {1, } ) );

            learning_rate_ /= ( 1.0 + decay_ * iterations_ );

            auto& ss = get_default_session<tensor_type>();//.get();
            for ( auto [id, v] :  ss.variables_ )
```

```
            {
                if (v.trainable_)
                {
                    auto& data = v.data();
                    auto& gradient = v.gradient();
                    auto& contexts = v.contexts();
                    if ( contexts.empty() ) // create context
                        contexts.push_back( zeros_like( data ) );
                        //contexts.push_back( std::make_shared<tensor_type>( zeros_like( data ) ) );
                    auto& moments = contexts[0];

                    for_each( moments.begin(), moments.end(), gradient.begin(), []( T& m, T g ) { m  += g*g; } );

                    for_each( data.begin(), data.end(), gradient.begin(), moments.begin(), [this]( T& d, T g, T m ) { d -= (*this).learning_rate_ * g / (eps + std::sqrt(m)); } );

                    gradient.reset(); // clear variable gradient
                }
            }
            ++iterations_;
        }//forward
    };//adagrad

    template< typename Loss, typename T >
    using ada_grad = adagrad<Loss, T>;

    template< typename Loss, typename T >
    struct rmsprop :  enable_id< rmsprop< Loss, T >, "rmsprop optimizer" >, enable_shared<rmsprop<Loss, T»
    {
        typedef tensor< T > tensor_type;

        Loss&          loss_;
        T              learning_rate_;
        T              rho_;
        T              decay_;
        unsigned long iterations_;

        rmsprop(Loss& loss, std::size_t batch_size, T learning_rate=1.0e-1, T rho=0.9, T decay=0.0) noexcept :
                loss_(loss), learning_rate_(learning_rate), rho_{rho},  decay_{std::max(T{0}, decay)}, iterations_{0}
        {
            better_assert( batch_size >= 1, "batch_size must be positive, but got:  ", batch_size );
            learning_rate_ /= static_cast<T>( batch_size );
        }

        void forward()
        {
            loss_.backward( ones<T>( {1, } ) );

            learning_rate_ /= ( 1.0 + decay_ * iterations_ );

            auto& ss = get_default_session<tensor_type>();//.get();
            for ( auto [id, v] :  ss.variables_ )
            {
                if (v.trainable_)
                {
                    auto& data = v.data();
                    auto& gradient = v.gradient();
                    auto& contexts = v.contexts();
                    if ( contexts.empty() ) // create context
                        contexts.push_back( zeros_like( data ) );
                        //contexts.push_back( std::make_shared<tensor_type>( zeros_like( data ) ) );
                    auto& moments = contexts[0];

                    if ( iterations_ == 0 )
                        for_each( moments.begin(), moments.end(), gradient.begin(), [this]( T& m, T g ) { m = g*g; } );
                    else
                        for_each( moments.begin(), moments.end(), gradient.begin(), [this]( T& m, T g ) { m *= (*this).rho_; m  += g*g*(1.0-(*this).rho_); } );

                    for_each( data.begin(), data.end(), gradient.begin(), moments.begin(), [this]( T& d, T g, T m ) { d -= (*this).learning_rate_ * g / (eps + std::sqrt(m)); } );

                    gradient.reset(); // clear variable gradient
                }
            }
            ++iterations_;
        }//forward
    };//rmsprop

    template< typename Loss, typename T >
    using rms_prop = rmsprop< Loss, T >;

```

```
    template< typename Loss, typename T >
    struct adadelta :  enable_id< adadelta< Loss, T >, "adadelta optimizer" >, enable_shared<adadelta<Loss, T>>
    {
        typedef tensor< T > tensor_type;

        Loss&          loss_;
        T              rho_;
        T              learning_rate_;
        unsigned long iterations_;

        adadelta(Loss& loss, std::size_t batch_size, T rho=0.9) noexcept :  loss_(loss), rho_{rho}, iterations_{0}
        {
            better_assert( batch_size >= 1, "batch_size must be positive, but got:  ", batch_size );
            learning_rate_ = T{1} / static_cast<T>( batch_size );
        }

        void forward()
        {
            loss_.backward( ones<T>( {1, } ) );

            auto& ss = get_default_session<tensor_type>();//.get();
            for ( auto [id, v] :  ss.variables_ )
            {
                if (v.trainable_)
                {
                    auto& data = v.data();
                    auto& gradient = v.gradient();
                    auto& contexts = v.contexts();
                    if ( contexts.empty() ) // create context
                    {
                        //contexts.push_back( std::make_shared<tensor_type>( zeros_like( data ) ) );
                        //contexts.push_back( std::make_shared<tensor_type>( zeros_like( data ) ) );
                        contexts.push_back( zeros_like( data ) );
                        contexts.push_back( zeros_like( data ) );
                    }
                    auto& moments = contexts[0];
                    auto& delta = contexts[0];

                    /*
if (iterations_==0)
{
for_each( moments.begin(), moments.end(), gradient.begin(), []( T& m, T g ) { m += g*g; } );
for_each( delta.begin(), delta.end(), gradient.begin(), []( T& d, T g ) { d += g*g; } );
}
else
{
// m = rho * m + (1-rho) * g * g;
for_each( moments.begin(), moments.end(), gradient.begin(), [this]( T& m, T g ) { m *= (*this).rho_; m += g*g*(1.0-(*this).rho_); } );
}
*/

                    for_each( moments.begin(), moments.end(), gradient.begin(), [this]( T& m, T g ) { m *= (*this).rho_; m  += g*g*(1.0-(*this).rho_); } );

                    // g_ = \sqrt{ (delta+eps) / (m+eps) }
                    for_each( gradient.begin(), gradient.end(), delta.begin(), moments.begin(), [this]( T& g, T d, T m ){ g *= (*this).learning_rate_ * std::sqrt((d+eps)/(m+eps));} );
                    // x = x - g_
                    data -= gradient;
                    // delta = rho * delta + (1-rho) * g_ * g_
                    /*
if (iterations_!=0)
*/
                    for_each( delta.begin(), delta.end(), gradient.begin(), [this]( T& d, T g ) { d *= (*this).rho_; d += (1.0-(*this).rho_) * g * g; } );

                    gradient.reset(); // clear variable gradient
                }
            }
            ++iterations_;
        }//forward
    };//adadelta

    template< typename Loss, typename T >
    using ada_delta = adadelta< Loss, T >;

    template< typename Loss, typename T >
    struct adam :  enable_id< adam< Loss, T >, "adam optimizer" >, enable_shared<adam<Loss, T>>
    {
        typedef tensor< T > tensor_type;

        Loss&          loss_;
        T              learning_rate_;
        T              beta_1_;
```

```
        T             beta_2_;
        bool          amsgrad_;
        unsigned long iterations_;

        adam(Loss& loss, std::size_t batch_size, T learning_rate=1.0e-1, T beta_1=0.9, T beta_2=0.999, bool amsgrad=false) noexcept :
             loss_{loss}, learning_rate_{learning_rate}, beta_1_{beta_1}, beta_2_{beta_2}, amsgrad_{ amsgrad }, iterations_{0}
        {
            better_assert( batch_size >= 1, "batch_size must be positive, but got:  ", batch_size );
            learning_rate_ /= static_cast<T>( batch_size );
        }

        void forward()
        {
            loss_.backward( ones<T>( {1, } ) );
            auto& ss = get_default_session<tensor_type>();//.get();
            for ( auto [id, v] :  ss.variables_ )
            {
                if (v.trainable_)
                {
                    auto& data = v.data();
                    auto& gradient = v.gradient();
                    auto& contexts = v.contexts();
                    if ( contexts.empty() ) // create context
                    {
                        //contexts.push_back( std::make_shared<tensor_type>( zeros_like( data ) ) );
                        //contexts.push_back( std::make_shared<tensor_type>( zeros_like( data ) ) );
                        contexts.push_back( zeros_like( data ) );
                        contexts.push_back( zeros_like( data ) );
                    }
                    auto& m = contexts[0];
                    auto& v = contexts[1];

                    T const b_beta_1 = beta_1_;
                    T const b_beta_2 = beta_2_;

                    for_each( m.begin(), m.end(), gradient.begin(), [b_beta_1](T& m_, T g_){ m_ *= b_beta_1; m_ += g_*(1.0-b_beta_1); } );

                    for_each( v.begin(), v.end(), gradient.begin(), [b_beta_2](T& v_, T g_){ v_ *= b_beta_2; v_ += g_* g_*(1.0-b_beta_2); } );

                    T lr = learning_rate_ * std::sqrt( 1.0 - std::pow(beta_2_, iterations_+1) ) / ( 1.0 - std::pow(beta_1_, iterations_+1) );

                    if ( iterations_ > 1 )
                        for_each( data.begin(), data.end(), m.begin(), v.begin(), [lr]( T& d_, T m_, T v_ ){ d_ -= lr * m_ / (eps+std::sqrt(v_)); } );
                    else
                        for_each( data.begin(), data.end(), gradient.begin(), [this]( T& d_, T g_ ){ d_ -= (*this).learning_rate_ * g_; } );

                    gradient.reset(); // clear variable gradient
                    // TODO: enabling amsgrad
                }
            }//loop of variables
            ++iterations_;
        }//adam::forward
    };// adam



    // Example usage:
    //
    //  //session ss;
    //  auto& ss = get_default_session<tensor<float»();
    //  auto loss = ...;
    //  auto optimizer = gradient{ loss, 1.0e-3f };
    //  for i = 1 :  1000
    //      ss.run( loss, batch_size )
    //      ss.run( optimizer )
    //
    template< typename Loss, typename T >
    struct gradient_descent :  enable_id< gradient_descent< Loss, T >, "gradient_descent optimizer" >, enable_shared<gradient_descent<Loss, T»
    {
        typedef tensor< T > tensor_type;
        Loss& loss_;
        T learning_rate_;
        T momentum_;

        gradient_descent(Loss& loss, std::size_t batch_size, T learning_rate=1.0e-3, T momentum=0.0) noexcept :  loss_(loss), learning_rate_(learning_rate), momentum_(momentum)
        {
            learning_rate_ /= static_cast<T>( batch_size ); // fix for batch size
        }
```

```

        void forward()
        {
            // update the gradient in the loss
            loss_.backward( ones<T>( {1, } ) );
            //update variables
            auto& ss = get_default_session<tensor_type>();//.get();
            for ( auto& [id, v] :  ss.variables_ )
            {
                if (v.trainable_)
                {
                    //v.data() -= learning_rate_ * (v.gradient());
                    //
                    auto& gradient = v.gradient();
                    better_assert( !has_nan(gradient), "gradient_descent error, tensor with id ", id, " has a nan value." );
                    v.data() -= learning_rate_ * gradient;
                    if (0)
                    {
                        std::ofstream ofs{ fmt::format("./debug/weight_{}.txt", id) };
                        ofs << v.gradient() << std::endl;
                        ofs.close();
                        better_assert( false, "stop here!" );
                    }

                    gradient.reset(); // clear variable gradient
                }
            }
        }

    };

    // TODO: adamax, nadam, ftrl



    //
    // optimizers interfaces
    //

    inline auto Adam = []( auto ...  args )
    {
        return [=]<Expression Ex>( Ex& loss )
        {
            return adam{loss, args...};
        };
    };

    inline auto SGD = []( auto ...  args )
    {
        return [=]<Expression Ex>( Ex& loss )
        {
            return sgd{loss, args...};
        };
    };

    inline auto Adagrad = []( auto ...  args )
    {
        return [=]<Expression Ex>( Ex& loss )
        {
            return adagrad{loss, args...};
        };
    };

    inline auto RMSprop = []( auto ...  args )
    {
        return [=]<Expression Ex>( Ex& loss )
        {
            return rmsprop{loss, args...};
        };
    };

    inline auto Adadelta = []( auto ...  args )
    {
        return [=]<Expression Ex>( Ex& loss )
        {
            return adadelta{loss, args...};
        };
    };


}//namespace ceras

#endif//XNRPSJMCYFXBDGNRJAWDNDIYQNGNXMRVLEHGNQWILKMTHGNOVHODLLXCCNIMUUFQSMOIYHDUD

```

9.27 /home/feng/workspace/github.repo/ceras/include/place_holder.hpp File Reference

```
#include "./includes.hpp"
#include "./tensor.hpp"
#include "./utils/better_assert.hpp"
#include "./utils/debug.hpp"
#include "./utils/id.hpp"
#include "./utils/enable_shared.hpp"
#include "./utils/state.hpp"
```

Classes

- struct ceras::place_holder_state< Tsor >
- struct ceras::place_holder< Tsor >
- struct ceras::is_place_holder< T >
- struct ceras::is_place_holder< place_holder< Tsor > >

Namespaces

- namespace ceras

Concepts

- concept ceras::Place_Holder

Functions

- template<Place_Holder Ph>
  bool ceras::operator== (Ph const &lhs, Ph const &rhs)
- template<Place_Holder Ph>
  bool ceras::operator!= (Ph const &lhs, Ph const &rhs)
- template<Place_Holder Ph>
  bool ceras::operator< (Ph const &lhs, Ph const &rhs)
- template<Place_Holder Ph>
  bool ceras::operator> (Ph const &lhs, Ph const &rhs)
- template<Place_Holder Ph>
  bool ceras::operator<= (Ph const &lhs, Ph const &rhs)
- template<Place_Holder Ph>
  bool ceras::operator>= (Ph const &lhs, Ph const &rhs)

Variables

- template<class T >
  constexpr bool ceras::is_place_holder_v = is_place_holder<T>::value

9.28 place_holder.hpp

Go to the documentation of this file.

```
#ifndef JPSBQEEADUPURARCCBVXOLXVMQHTNWCQWXUKKHCOTWFGOGXSODKEEYLSSTFGTVXNBROLKKEJM
#define JPSBQEEADUPURARCCBVXOLXVMQHTNWCQWXUKKHCOTWFGOGXSODKEEYLSSTFGTVXNBROLKKEJM

#include "./includes.hpp"
#include "./tensor.hpp"
#include "./utils/better_assert.hpp"
#include "./utils/debug.hpp"
#include "./utils/id.hpp"
#include "./utils/enable_shared.hpp"
#include "./utils/state.hpp"

namespace ceras
{

    template< Tensor Tsor >
    struct place_holder_state
    {
        Tsor data_;
        std::vector< unsigned long> shape_hint_;
    };

    template< Tensor Tsor >
    struct place_holder :  enable_id< place_holder<Tsor>, "PlaceHolder" >,
   enable_shared_state<place_holder<Tsor>, place_holder_state<Tsor>»
    {
        typedef Tsor tensor_type;

        place_holder( place_holder const& other) = default;
        place_holder( place_holder && other) = default;
        place_holder& operator = ( place_holder const& other) = default;
        place_holder& operator = ( place_holder && other) = default;

        place_holder()
        {
            (*this).state_ = std::make_shared<place_holder_state<Tsor»();
            (*((*this).state_)).shape_hint_ = std::vector< unsigned long >{ {-1UL,} };
        }

        place_holder( std::vector<unsigned long> const& shape_hint )
        {
            (*this).state_ = std::make_shared<place_holder_state<Tsor»();
            (*((*this).state_)).shape_hint_ = shape_hint;
            if ( shape_hint[0] != -1UL )
            {
                auto & si = (*((*this).state_)).shape_hint_;
                si.insert( si.begin(), 1UL );
            }
        }

        void bind( Tsor data )
        {
            better_assert( (*this).state_, "Error with empty state." );
            (*((*this).state_)).data_ = data;
            (*((*this).state_)).shape_hint_ = data.shape();
        }

        Tsor const forward()const
{
            better_assert( (*this).state_, "Error with empty state." );
            better_assert( !((*((*this).state_)).data_.empty()), "Error with empty tensor." );
            return (*((*this).state_)).data_;
        }

        void reset() noexcept
        {
            (*((*this).state_)).data_ = Tsor{};
            (*((*this).state_)).shape_hint_ = std::vector<unsigned long>{{-1UL}};
        }

        void backward( auto ) const noexcept { }

        void shape( std::vector< unsigned long> const& shape_hint ) noexcept
        {
            (*((*this).state_)).shape_hint_ = shape_hint;
        }

        std::vector<unsigned long> shape() const noexcept
        {
            if ( !  (*((*this).state_)).data_.empty() )
                return (*((*this).state_)).data_.shape();

            //debug_log( fmt::format("calculating the shape for place_holder with id {} ", (*this).id())
   );
```

```
            //debug_log( fmt::format("got {} ",  (*((*this).state_)).shape_hint_ ) ) ;
            return (*((*this).state_)).shape_hint_;
        }

    };

    template< typename T >
    struct is_place_holder :  std::false_type {};

    template< Tensor Tsor >
    struct is_place_holder< place_holder< Tsor > > :  std::true_type {};

    template< class T >
    inline constexpr bool is_place_holder_v = is_place_holder<T>::value;

    template< typename T >
    concept Place_Holder = is_place_holder_v<T>;

    template< Place_Holder Ph >
    bool operator == ( Ph const& lhs, Ph const& rhs )
    {
        return lhs.id() == rhs.id();
    }

    template< Place_Holder Ph >
    bool operator != ( Ph const& lhs, Ph const& rhs )
    {
        return lhs.id() != rhs.id();
    }

    template< Place_Holder Ph >
    bool operator < ( Ph const& lhs, Ph const& rhs )
    {
        return lhs.id() < rhs.id();
    }

    template< Place_Holder Ph >
    bool operator > ( Ph const& lhs, Ph const& rhs )
    {
        return lhs.id() < rhs.id();
    }

    template< Place_Holder Ph >
    bool operator <= ( Ph const& lhs, Ph const& rhs )
    {
        return lhs.id() <= rhs.id();
    }

    template< Place_Holder Ph >
    bool operator >= ( Ph const& lhs, Ph const& rhs )
    {
        return lhs.id() >= rhs.id();
    }

}//namespace ceras

#endif//JPSBQEEADUPURARCCBVXOLXVMQHTNWCQWXUKKHCOTWFGOGXSODKEEYLSSTFGTVXNBROLKKEJM

```

9.29 /home/feng/workspace/github.repo/ceras/include/recurrent.hpp File Reference

```
#include "./constant.hpp"
#include "./session.hpp"
#include "./operation.hpp"
#include "./activation.hpp"
#include "./layer.hpp"
```

Namespaces

- namespace ceras

Functions

- auto ceras::lstm (std::unsigned long units) noexcept

9.29.1 Variable Documentation

9.29.1.1 units_

```
unsigned long units_
```

9.30 recurrent.hpp

Go to the documentation of this file.

```
#ifndef QGIVVEUESCDFTUPFLDJSNBGBWHLCNAILBAKEBNCOGGOBHFQUQOTGENHFEHKQDGCDVWQSMJOQL
#define QGIVVEUESCDFTUPFLDJSNBGBWHLCNAILBAKEBNCOGGOBHFQUQOTGENHFEHKQDGCDVWQSMJOQL

#include "./constant.hpp"
#include "./session.hpp"
#include "./operation.hpp"
#include "./activation.hpp"
#include "./layer.hpp"

namespace ceras
{

    namespace
    {
        struct lstm_context
        {
            unsigned long units_;
        };



        template< Expression Ex, Expression Ey >
        auto copy_state( Ex const& ex, Ey const& ey ) noexcept
        {
            std::shared_ptr<std::any> forward_cache = std::make_shared<std::any>();
            return make_binary_operator( [forward_cache]<Tensor Tsor>( Tsor const&, Tsor const& rhs_tensor ) noexcept
                                         {
                                            Tsor& ans = context_cast<Tsor>( forward_cache );
                                            ans.resize( rhs_tensor.shape() ); // note:  when batch_size differes, the shape migh change
                                            std::copy( rhs_tensor.begin(), rhs_tensor.end(), ans.begin() );
                                            return ans;
                                         },
                                         []<Tensor Tsor>( Tsor const& lhs_input, Tsor const& rhs_input, Tsor const&, Tsor const& ) noexcept
                                         {
                                            auto z = zeros_like( lhs_input );
                                            return std::make_tuple( z, z );
                                         },
                                         "CopyState"
                    )( lhs_ex, rhs_ex );
        }

    }//anonymous namespace


    inline auto lstm( std::unsigned long units ) noexcept
    {
        std::shared_ptr<std::any> short_term_memory = std::make_shared<std::any>();
        std::shared_ptr<std::any> long_term_memory = std::make_shared<std::any>();

        return [=]<Expression Ex>( Ex const& ex ) noexcept
        {
            variable h_; // previous h
```

```
            varialbe c_; // previous c

            auto hx = concatenate( -1 )( h_, ex );
            auto f = sigmoid( Dense( units, units+units )( hx ) );
            auto i = sigmoid( Dense( units, units+units )( hx ) );
            auto ca = tanh( Dense( units, units+units )( hx ) );
            auto c = hadamard_product( f, c_ ) + hadamard_product( i, ca );

            auto o = sigmoid( Dense( units, units+units )( hx ) );
            auto h = hadamard_product( o, tanh( c ) );

            auto reserve_h = zeros_like( assign( h_, h ) );
            auto reserve_c = zeros_like( assign( c_, c ) );

            return o + reserve_h + reserve_c; // 'reserve_h' and 'reserve_c' are zeros.  They are here
    just to prevent 'reserve_h' and 'reserve_c' from being optimized out.
        };
    };




}//namespace ceras

#endif//QGIVVEUESCDFTUPFLDJSNBGBWHLCNAILBAKEBNCOGGOBHFQUQOTGENHFEHKQDGCDVWQSMJOQL

```

9.31 /home/feng/workspace/github.repo/ceras/include/session.hpp File Reference

```
#include "./includes.hpp"
#include "./tensor.hpp"
#include "./place_holder.hpp"
#include "./variable.hpp"
#include "./utils/singleton.hpp"
#include "./utils/debug.hpp"
#include "./utils/lzw.hpp"
#include "./utils/fmt.hpp"
```

Classes

- struct ceras::ceras_private::session< Tsor >

Namespaces

- namespace ceras
- namespace ceras::ceras_private

Functions

- template<Tensor Tsor>
  ceras_private::session< Tsor > & ceras::get_default_session ()

  Get the default global session.
- template<Tensor Tsor>
  auto & ceras::bind (place_holder< Tsor > &p_holder, Tsor const &value)

  Bind a tensor to a place holder.
- template<typename Operation >
  auto ceras::run (Operation &op)

  Run an expression.

9.32 session.hpp

Go to the documentation of this file.

```
#ifndef NRFLVKIAQLDTRLNHHBYUJJAMYCRCFKLQTDSKDQSALHQGURGGKBSIGGVWXBSKHQGPAUDLPUBBQ
#define NRFLVKIAQLDTRLNHHBYUJJAMYCRCFKLQTDSKDQSALHQGURGGKBSIGGVWXBSKHQGPAUDLPUBBQ

#include "./includes.hpp"
#include "./tensor.hpp"
#include "./place_holder.hpp"
#include "./variable.hpp"
#include "./utils/singleton.hpp"
#include "./utils/debug.hpp"
#include "./utils/lzw.hpp"
#include "./utils/fmt.hpp"

namespace ceras
{

    namespace ceras_private
    {

    template< Tensor Tsor >
    struct session
    {
        typedef place_holder<Tsor> place_holder_type;
        typedef variable<Tsor> variable_type;
        typedef variable_state<Tsor> variable_state_type;

        std::vector<place_holder_type> place_holders_;
        std::map<int, variable_type> variables_;

        session() { }

        session( session const& ) = delete;
        session( session&& ) = default;
        session& operator=( session const& ) = delete;
        session& operator=( session&& ) = default;

        session& rebind( place_holder_type& p_holder, Tsor const& value )
        {
            p_holder.bind( value );
            return *this;
        }

        session& bind( place_holder_type& p_holder, Tsor const& value )
        {
            p_holder.bind( value );
            place_holders_.emplace_back( p_holder );
            return *this;
        }

        session& remember( variable_type const& v )
        {
            if ( variables_.find( v.id_ ) == variables_.end() )
                variables_.insert( {v.id_, v} );
            return *this;
        }

        template< typename Operation >
        auto run( Operation& op )const
{
            return op.forward();
        }

        // register variables associated to the op to this session
        // usually being called before restoring a session from a file
        template< typename Operation >
        void tap( Operation& op )const
{
            run( op );
        }

        void deserialize( std::string const& file_path )
        {
            restore( file_path );
        }

        void serialize( std::string const& file_path )const
{
            save( file_path );
        }

        void save( std::string const& file_path )const
{
            // find a tmp file
```

```
            std::string const& tmp_file_path =  file_path + std::string{".tmp"};

            // save original to tmp file
            save_original( tmp_file_path );

            // compress tmp file to file_path
            {
                std::ifstream ifs{ tmp_file_path, std::ios_base::binary };
                std::ofstream ofs( file_path, std::ios_base::binary );
                lzw::compress( ifs, ofs );
            }

            // remove original
            std::remove( tmp_file_path.c_str() );
        }

        void restore( std::string const& file_path )
        {
            // find a tmp file
            std::string const& tmp_file_path =  file_path + std::string{".tmp"};

            // uncompress tmp file
            {
                std::ifstream ifs( file_path, std::ios_base::binary );
                std::ofstream ofs{ tmp_file_path, std::ios_base::binary };
                lzw::decompress( ifs, ofs );
            }

            // restore original from tmp file to file_path
            restore_original( tmp_file_path );

            // remove tmp file
            //std::remove( tmp_file_path );
            std::remove( tmp_file_path.c_str() );
        }

        void save_original( std::string const& file_path )const
{
            std::ofstream ofs{ file_path };
            better_assert( ofs.good(), "failed to open file ", file_path );

            // save id
            for ( auto const& [id, v] :  variables_ )
            {
                ofs « id « " ";
            }
            ofs « "\n";

            // save tensors
            for ( auto const& [id, v] :  variables_ )
            {
                write_tensor( ofs, v.data() );
            }

            ofs.close();
        }

        void restore_original( std::string const& file_path )
        {
            std::ifstream ifs{ file_path };
            better_assert( ifs.good(), "failed to open file ", file_path );

            // get list of ids from the 1st line
            std::vector<int> ids;
            {
                std::string str_ids;
                std::getline( ifs, str_ids );
                std::stringstream ss( str_ids );
                std::copy( std::istream_iterator<int>( ss ), std::istream_iterator<int>(),
    std::back_inserter( ids ) );
            }

            // restore each of the tensor, ignoring their gradients
            for ( auto id :  ids )
            {
                auto itor = variables_.find( id );
                better_assert( itor != variables_.end(), "Error:  unknown variable to load, the id is ",
    id );

                auto [_id, _var] = *itor;
                read_tensor( ifs, _var.data() );
            }

            ifs.close();
        }

        ~session()
```

```
        {
            for ( auto& p_holder :  place_holders_ )
                p_holder.reset();

            place_holders_.clear();
            variables_.clear();

            singleton<session<Tsor>*>::instance() = nullptr;
        }
    }; // session

    } //namespace ceras_private

    template< Tensor Tsor >
    ceras_private::session<Tsor>& get_default_session()
    {
        return singleton<ceras_private::session<Tsor»::instance();
    }


    template< Tensor Tsor >
    auto& bind( place_holder<Tsor>& p_holder, Tsor const& value )
    {
        auto& ss = get_default_session<Tsor>();
        ss.bind( p_holder, value );
        return ss;
    }

    template< typename Operation >
    auto run( Operation& op )
    {
        typedef typename Operation::tensor_type tensor_type;
        auto ss = get_default_session<tensor_type>();
        return ss.run( op );
    }

}//namespace ceras

#endif//NRFLVKIAQLDTRLNHHBYUJJAMYCRCFKLQTDSKDQSALHQGURGGKBSIGGVWXBSKHQGPAUDLPUBBQ

```

9.33 /home/feng/workspace/github.repo/ceras/include/tensor.hpp File Reference

```
#include "./backend/cblas.hpp"
#include "./backend/cuda.hpp"
#include "./config.hpp"
#include "./includes.hpp"
#include "./utils/better_assert.hpp"
#include "./utils/buffered_allocator.hpp"
#include "./utils/debug.hpp"
#include "./utils/fmt.hpp"
#include "./utils/for_each.hpp"
#include "./utils/id.hpp"
#include "./utils/range.hpp"
#include "./utils/stride_iterator.hpp"
#include "./utils/view.hpp"
#include "./tensor.tcc"
```

Classes

- struct ceras::tensor< T, Allocator >
- struct ceras::is_tensor< T >
- struct ceras::is_tensor< tensor< T, A > >

Namespaces

- namespace ceras

Concepts

- concept ceras::Tensor

Typedefs

- template<typename T >
  using ceras::default_allocator = buffered_allocator< T, 256 >

Functions

- template<typename T , typename A = default_allocator<T>>
  constexpr tensor< T, A > ceras::as_tensor (T val) noexcept

Variables

- static unsigned long ceras::random_seed = std::chrono::system_clock::now().time_since_epoch().count()

  Random seed for the tensor library.
- static std::mt19937 ceras::random_generator {random_seed}
- template<class T >
  constexpr bool ceras::is_tensor_v = is_tensor<T>::value

9.34 tensor.hpp

Go to the documentation of this file.

```
#ifndef HQKGLAXWWVFBFHQNHBVTQJKGUFTPCQPTPXDVNOSBDJIBHITCEKDISJYNAMCPLJDURURDAISFV
#define HQKGLAXWWVFBFHQNHBVTQJKGUFTPCQPTPXDVNOSBDJIBHITCEKDISJYNAMCPLJDURURDAISFV

#include "./backend/cblas.hpp"
#include "./backend/cuda.hpp"
#include "./config.hpp"
#include "./includes.hpp"
#include "./utils/better_assert.hpp"
#include "./utils/buffered_allocator.hpp"
#include "./utils/debug.hpp"
#include "./utils/fmt.hpp"
#include "./utils/for_each.hpp"
#include "./utils/id.hpp"
#include "./utils/range.hpp"
#include "./utils/stride_iterator.hpp"
#include "./utils/view.hpp"

namespace ceras
{
    static unsigned long random_seed = std::chrono::system_clock::now().time_since_epoch().count();

    // static random number random_generator
    static std::mt19937 random_generator{random_seed};

    template< typename T >
    using default_allocator = buffered_allocator<T, 256>;
    //using default_allocator = std::allocator<T>;


    template< typename T, typename Allocator = default_allocator<T> >
    struct tensor :  enable_id<tensor<T, Allocator>, "Tensor">
    {
```

```
        typedef T value_type;
        typedef Allocator allocator;
        typedef std::vector<T, Allocator> vector_type;
        typedef std::shared_ptr<vector_type> shared_vector;
        typedef tensor self_type;

        std::vector<unsigned long> shape_;
        unsigned long memory_offset_;
        shared_vector vector_;

        tensor() :  shape_{std::vector<unsigned long>{}}, memory_offset_{0},
    vector_{std::make_shared<vector_type>()} { }

        constexpr tensor( std::vector<unsigned long> const& shape, std::initializer_list<T> init, const
    Allocator& alloc = Allocator() ) :  shape_{shape}, memory_offset_{0},
    vector_{std::make_shared<vector_type>(init, alloc)}
        {
            better_assert( (*vector_).size() == std::accumulate( shape_.begin(), shape_.end(), 1UL,
    [](auto x, auto y){ return x*y; } ), "Expecting vector has same size as the shape indicates." );
        }

        constexpr tensor( std::vector<unsigned long> const& shape ) :  shape_{shape}, memory_offset_{0},

    vector_{std::make_shared<vector_type>(std::accumulate(shape_.begin(), shape_.end(), 1UL, [](auto x,
    auto y){return x*y;} ), T{0})}{}

        constexpr tensor( std::vector<unsigned long> const& shape, T init ) :  shape_{shape},
    memory_offset_{0},

    vector_{std::make_shared<vector_type>(std::accumulate(shape_.begin(), shape_.end(), 1UL, [](auto x,
    auto y){return x*y;}), T{0})}
        {
            std::fill( begin(), end(), init );
        }

        constexpr tensor( tensor const& other, unsigned long memory_offset ) noexcept :  shape_{
    other.shape_ }, memory_offset_{ memory_offset }, vector_{ other.vector_ } {}

        constexpr tensor( self_type const& other ) noexcept :  shape_{ other.shape_ }, memory_offset_{
    other.memory_offset_ }
        {
            vector_ = other.vector_;
            (*this).id_ = other.id_;
        }

        constexpr tensor( self_type && other ) noexcept :  shape_{ other.shape_ }, memory_offset_{
    other.memory_offset_ }
        {
            vector_ = other.vector_;
            (*this).id_ = other.id_;
        }

         constexpr self_type& operator = ( self_type const& other ) noexcept
         {
             shape_ = other.shape_;
             memory_offset_ = other.memory_offset_;
             vector_ = other.vector_;
             (*this).id_ = other.id_;
             return *this;
         }

         constexpr self_type& operator = ( self_type && other ) noexcept
         {
             shape_ = other.shape_;
             memory_offset_ = other.memory_offset_;
             vector_ = other.vector_;
             (*this).id_ = other.id_;
             return *this;
         }

         constexpr auto begin() noexcept
         {
             return data();
         }

         constexpr auto begin() const noexcept
         {
             return data();
         }

         constexpr auto cbegin() const noexcept
         {
             return begin();
         }

         constexpr auto end() noexcept
         {
```

```
            return begin() + size();
        }

        constexpr auto end() const noexcept
        {
            return begin() + size();
        }

        constexpr auto cend() const noexcept
        {
            return  end();
        }


        constexpr auto rbegin() noexcept
        {
            return make_reverse_iterator( end() );
        }

        constexpr auto rbegin() const noexcept
        {
            return make_reverse_iterator( end() );
        }

        constexpr auto crbegin() const noexcept
        {
            return make_reverse_iterator( cend() );
        }

        constexpr auto rend() noexcept
        {
            return make_reverse_iterator( begin() );
        }

        constexpr auto rend() const noexcept
        {
            return make_reverse_iterator( begin() );
        }

        constexpr auto crend() const noexcept
        {
            return make_reverse_iterator( cbegin() );
        }

        constexpr auto front()
        {
            return (*vector_).front();
        }

        constexpr auto front()const
{
            return (*vector_).front();
        }


        constexpr auto back()
        {
            return (*vector_).back();
        }

        constexpr auto back()const
{
            return (*vector_).back();
        }


        constexpr unsigned long size() const noexcept
        {
            if ( !vector_ ) return 0;
            return (*vector_ ).size() - memory_offset_;
            //return std::accumulate( shape_.begin(), shape_.end(), 1UL, [](unsigned long x, unsigned long y){return x*y;} );
        }



        [[nodiscard]] constexpr bool empty() const noexcept
        {
            return cbegin() == cend();
        }



        constexpr self_type& reset( T val = T{0} )
        {
            std::fill_n( data(), size(), val );
            return *this;
        }
```

```
        constexpr unsigned long ndim() const noexcept
        {
            return shape_.size();
        }

        constexpr std::vector<unsigned long> const& shape() const noexcept
        {
            return shape_;
        }


        constexpr self_type& deep_copy( self_type const& other )
        {
            (*this).resize( other.shape() );
            std::copy_n( other.data(), size(), (*this).data() );
            return *this;
        }

        constexpr self_type const deep_copy()const
{
            self_type ans{ shape_ };
            std::copy_n( data(), size(), ans.data() );
            return ans;
        }

        constexpr self_type const copy()const
{
            return deep_copy();
        }

        // 1-D view
        constexpr value_type& operator[]( unsigned long idx )
        {
            return *(data()+idx);
        }

        // 1-D view
        constexpr value_type const& operator[]( unsigned long idx )const
{
            return *(data()+idx);
        }

        constexpr self_type& resize( std::vector< unsigned long > const& new_shape )
        {
            unsigned long const new_size = std::accumulate( new_shape.begin(), new_shape.end(), 1UL, [](auto x, auto y){ return x*y; } );
            if( (*this).size() != new_size )
            {
                (*vector_).resize(new_size);
                memory_offset_ = 0UL;
            }
            (*this).shape_ = new_shape;
            return *this;
        }

        constexpr self_type& reshape( std::vector<unsigned long> const& new_shape )
        {
            std::vector<unsigned long> _new_shape = new_shape;
            if ( *(_new_shape.rbegin()) == static_cast<unsigned long>( -1 ) )
                *(_new_shape.rbegin()) = (*this).size() / std::accumulate( _new_shape.begin(), _new_shape.end()-1, 1Ul, []( unsigned long x, unsigned long y ){ return x*y; } );

            unsigned long const new_size = std::accumulate( _new_shape.begin(), _new_shape.end(), 1UL, [](auto x, auto y){ return x*y; } );
            if ( (*this).size() != new_size ) return resize( _new_shape );

            better_assert( (*this).size() == new_size, "reshape:  expecting same size, but the original size is ", (*this).size(), ", and the new size is ", new_size );
            (*this).shape_ = _new_shape;
            return *this;
        }

        //mapping a smaller tensor on a larger one
        constexpr self_type& shrink_to( std::vector< unsigned long > const& new_shape )
        {
            unsigned long const new_size = std::accumulate( new_shape.begin(), new_shape.end(), 1UL, [](auto x, auto y){ return x*y; } );
            better_assert( (*this).size() >= new_size, "reshape:  expecting smaller size, but the original size is ", (*this).size(), ", and the new size is ", new_size );
            (*this).shape_ = new_shape;
            return *this;
        }

        //adjust the memory offset
        constexpr self_type& creep_to( unsigned long new_memory_offset )
        {
```

```
            (*this).memory_offset_ = new_memory_offset;
            return *this;
        }

        constexpr value_type* data() noexcept
        {
            return (*vector_).data() + memory_offset_;
        }

        constexpr const value_type* data() const noexcept
        {
            return (*vector_).data() + memory_offset_;
        }

        template< typename Function >
        constexpr self_type& map( Function const& f )
        {
            for_each( (*this).data(), (*this).data()+(*this).size(), [&f]( auto& v ){ f(v); } );
            return *this;
        }

        constexpr self_type& operator += ( self_type const& other )
        {
            //better_assert( shape() == other.shape(), "Error with tensor::operator += :  Shape mismatch!  -- current shape is ", shape(), " and other tensor shape is ", other.shape() );
            better_assert( shape() == other.shape(), fmt::format("Error with tensor::operator += : Shape mismatch!  This shape is {}, while other shape is {}.", shape(), other.shape() ) );
            std::transform( data(), data()+size(), other.data(), data(), []( auto x, auto y ){ return x+y; } );
            return *this;
        }

        constexpr self_type& operator += ( value_type x )
        {
            for_each( data(), data()+size(), [x]( value_type& v ){ v += x; } );
            return *this;
        }

        constexpr self_type& operator -= ( self_type const& other )
        {
            better_assert( shape() == other.shape(), "Error with tensor::operator -=:  Shape not match!" );
            std::transform( data(), data()+size(), other.data(), data(), []( auto x, auto y ){ return x-y; } );
            return *this;
        }

        constexpr self_type& operator -= ( value_type x )
        {
            for_each( data(), data()+size(), [x]( auto& v ){ v -= x; } );
            return *this;
        }

        constexpr self_type& operator *= ( self_type const& other )
        {
            better_assert( shape() == other.shape(), "Shape not match!" );
            std::transform( data(), data()+size(), other.data(), data(), []( auto x, auto y ){ return x*y; } );
            return *this;
        }

        constexpr self_type& operator *= ( value_type x )
        {
            for_each( data(), data()+size(), [x]( auto& v ){ v *= x; } );
            return *this;
        }

        constexpr self_type& operator /= ( self_type const& other )
        {
            better_assert( shape() == other.shape(), "Shape not match!" );
            std::transform( data(), data()+size(), other.data(), data(), []( auto x, auto y ){ return x/y; } );
            return *this;
        }

        constexpr self_type& operator /= ( value_type x )
        {
            for_each( data(), data()+size(), [x]( auto& v ){ v /= x; } );
            return *this;
        }

        constexpr self_type const operator - ()const
{
            self_type ans = (*this).deep_copy();
            for_each( ans.data(), ans.data()+size(), []( auto& v ){ v = -v; } );
            return  ans;
        }
```

```
        constexpr value_type as_scalar() const noexcept
        {
            better_assert( size() == 1, "Expecting tensor has a single value, but got ", size() );
            return *begin();
        }

        template< typename U >
        constexpr auto as_type() const noexcept
        {
            tensor<U, typename std::allocator_traits<Allocator>::rebind_alloc<U>> ans{ (*this).shape() };
            std::copy( (*this).begin(), (*this).end(), ans.begin() );
            return ans;
        }

        tensor slice( unsigned long m, unsigned long n ) const noexcept
        {
            better_assert( m < n, "starting dimension larger than then ending dimension." );
            better_assert( !shape_.empty(), "Cannot slice an empty tensor." );

            unsigned long first_dim = shape_[0];
            better_assert( n <= first_dim, "this tensor only has ", first_dim, " at the first dimension, too small for n = ", n );

            unsigned long rest_dims = std::accumulate( shape_.begin()+1, shape_.end(), 1UL, []( auto x, auto y ){ return x*y; } );

            tensor ans = *this;
            ans.shape_[0] = n - m;
            ans.memory_offset_ = rest_dims * m + memory_offset_;
            return ans;
        }

    };

    template <typename T, typename A=default_allocator<T> >
    constexpr tensor<T, A> as_tensor( T val ) noexcept
    {
        tensor<T, A> ans{ {1,} };
        ans[0] = val;
        return ans;
    }

    template< typename T >
    struct is_tensor :  std::false_type {};

    template< typename T, typename A >
    struct is_tensor< tensor< T, A> > :  std::true_type {};

    template< class T >
    inline constexpr bool is_tensor_v = is_tensor<T>::value;

    template< typename T >
    concept Tensor = is_tensor_v<T>;


}//namespace ceras

#include "./tensor.tcc"

#endif//HQKGLAXWWVFBFHQNHBVTQJKGUFTPCQPTPXDVNOSBDJIBHITCEKDISJYNAMCPLJDURURDAISFV
```

9.35 /home/feng/workspace/github.repo/ceras/include/value.hpp File Reference

```
#include "./includes.hpp"
#include "./tensor.hpp"
#include "./utils/id.hpp"
#include "./utils/better_assert.hpp"
#include "./utils/enable_shared.hpp"
```

Classes

- struct ceras::value< T >

- struct ceras::is_value< T >
- struct ceras::is_value< value< T > >
- struct ceras::tensor_deduction< L, R >

Namespaces

- namespace ceras

Concepts

- concept ceras::Value

Variables

- template<class T >
  constexpr bool ceras::is_value_v = is_value<T>::value

9.36 value.hpp

Go to the documentation of this file.

```
#ifndef VALUE_HPP_INCLUDED_DS9P8IU4LKJASDOIPUY498YAFKASHFAS9F8Y4OKHDAFSIUOHASDFFS
#define VALUE_HPP_INCLUDED_DS9P8IU4LKJASDOIPUY498YAFKASHFAS9F8Y4OKHDAFSIUOHASDFFS

#include "./includes.hpp"
#include "./tensor.hpp"
#include "./utils/id.hpp"
#include "./utils/better_assert.hpp"
#include "./utils/enable_shared.hpp"

namespace ceras
{

    template< typename T > requires std::floating_point<T>
    struct value :  enable_id< value<T>, "Value" >
    {
        typedef T value_type;
        typedef tensor<value_type> tensor_type;
        value_type data_;

        value() = delete;
        value( value_type v ) noexcept :  enable_id<value<T>, "Value">{}, data_{ v } {}
        value( value const& ) noexcept = default;
        value( value && ) noexcept = default;
        value& operator =( value const& ) noexcept = default;
        value& operator =( value && ) noexcept = default;

        void backward( auto ) noexcept { }

        template< Tensor Tsor >
        Tsor const forward( Tsor const& refer )const
{
            Tsor ans = ones_like( refer ); // cast it to a tensor
            ans *= data_;
            return ans;
        }

        std::vector<unsigned long> shape() const noexcept
        {
            return std::vector<unsigned long>{ {-1UL,} };
        }

    };//struct value

    template< typename T >
    struct is_value :  std::false_type {};

    template< typename T >
```

```
    struct is_value< value< T > > :  std::true_type {};

    template< class T >
    inline constexpr bool is_value_v = is_value<T>::value;

    template< typename T >
    concept Value = is_value_v<T>;


    // for tensor_type deduction in a binary operator
    template< typename L, typename R >
    struct tensor_deduction
    {
        using op_type = std::conditional<is_value_v<L>, R, L>::type;
        using tensor_type = std::remove_cv_t<decltype(std::declval<op_type>().forward())>;
    };


}//namespace ceras

#endif//VALUE_HPP_INCLUDED_DS9P8IU4LKJASDOIPUY498YAFKASHFAS9F8Y4OKHDAFSIUOHASDFFS

```

9.37 /home/feng/workspace/github.repo/ceras/include/variable.hpp File Reference

```
#include "./includes.hpp"
#include "./tensor.hpp"
#include "./utils/id.hpp"
#include "./utils/debug.hpp"
#include "./config.hpp"
#include "./utils/enable_shared.hpp"
#include "./utils/state.hpp"
```

Classes

- struct ceras::variable_state< Tsor >
- struct ceras::regularizer< Float >
- struct ceras::variable< Tsor >
- struct ceras::is_variable< T >
- struct ceras::is_variable< variable< Tsor > >

Namespaces

- namespace ceras
- namespace ceras::ceras_private

Concepts

- concept ceras::Variable

Functions

- template<Tensor Tsor>
  ceras_private::session< Tsor > & ceras::get_default_session ()

  Get the default global session.
- template<Variable Var>
  bool ceras::operator== (Var const &lhs, Var const &rhs) noexcept

Variables

- template<class T >
  constexpr bool ceras::is_variable_v = is_variable<T>::value

9.38 variable.hpp

Go to the documentation of this file.

```
#ifndef QVETFVLYKDJJLDBPAMVBUWUGPWXIAIGMXUDVOFGQIHUHVOTBAWEMPJQEWJQIGGSTUCNDHLUYL
#define QVETFVLYKDJJLDBPAMVBUWUGPWXIAIGMXUDVOFGQIHUHVOTBAWEMPJQEWJQIGGSTUCNDHLUYL

#include "./includes.hpp"
#include "./tensor.hpp"
#include "./utils/id.hpp"
#include "./utils/debug.hpp"
#include "./config.hpp"
#include "./utils/enable_shared.hpp"
#include "./utils/state.hpp"

namespace ceras
{

    namespace ceras_private
    {
        template< Tensor Tsor >
        struct session;
    }

    template< Tensor Tsor >
    ceras_private::session<Tsor>& get_default_session();

    template< Tensor Tsor >
    struct variable_state
    {
        Tsor data_;
        Tsor gradient_;
        std::vector<Tsor> contexts_;
    };

    template< typename Float > requires std::floating_point<Float>
    struct regularizer
    {
        typedef Float value_type;
        value_type l1_;
        value_type l2_;
        bool synchronized_;

        constexpr regularizer( value_type l1=0.0, value_type l2=0.0, bool synchronized=false ) noexcept : l1_{l1}, l2_{l2}, synchronized_{synchronized} {}
    };

    template< Tensor Tsor >
    struct variable :  enable_id<variable<Tsor>, "Variable">
    {
        typedef Tsor tensor_type;
        typedef typename tensor_type::value_type value_type;

        std::shared_ptr<variable_state<tensor_type» state_;
        regularizer<value_type> regularizer_;
        bool trainable_;

        variable( tensor_type const& data, value_type l1=value_type{0}, value_type l2=value_type{0}, bool trainable=true ) :  enable_id<variable<tensor_type>, "Variable">{}, regularizer_{l1, l2, true}, trainable_{trainable}
        {
            (*this).state_ = std::make_shared<variable_state<tensor_type»();
            (*((*this).state_)).data_ = data;
            (*((*this).state_)).gradient_ = tensor_type{ data.shape() };

            auto& ss = get_default_session<tensor_type>();
            ss.remember( *this );
        }

        //variable() = delete;
        variable() noexcept {}
        variable( variable const& other ) = default;
        variable( variable && ) = default;
        variable& operator=( variable&&) = default;
        variable& operator=( variable const& other) = default;

```

```
        tensor_type const forward() noexcept// const
        {
            auto& state = *((*this).state_);

            if ( learning_phase == 1 )
            {
                typedef typename tensor_type::value_type value_type;
                state.gradient_.reset( value_type{0} );
                regularizer_.synchronized_ = false; // mark changes
            }
            return state.data_;
        }

        void backward( auto const& grad ) noexcept
        {
            if (!trainable_) return;

            auto& state = *((*this).state_);
            {
                if (state.gradient_.shape() != state.data_.shape())
                    state.gradient_.resize( state.data_.shape() );
            }
            state.gradient_ += grad; // collecting all the gradients from its children nodes, will be called mulitple times in a single backward pass

            // apply regularizers
            if (!(regularizer_.synchronized_)) // in case of multiple invoke of this method in a same backward pass
            {
                if ( regularizer_.l1_ >= eps ) // l1 regularizer
                {
                    value_type const factor = regularizer_.l1_;
                     for_each( state.data_.begin(), state.data_.end(), state.gradient_.begin(), [factor]( value_type d, value_type& g ){ g += (d >= value_type{0}) ?  factor :  -factor; } );
                 }
                 if ( regularizer_.l2_ >= eps ) // l2 regularizer
                 {
                     value_type const factor = regularizer_.l2_;
                     for_each( state.data_.begin(), state.data_.end(), state.gradient_.begin(), [factor]( value_type d, value_type& g ){ g += value_type{2} * d * factor; } );
                 }

                 regularizer_.synchronized_ = true;
             }
         }

         std::vector<std::size_t> shape() const noexcept
         {
             auto& state = *((*this).state_);
             //debug_log( fmt::format("calculating the shape of variable with id {}, got {}", (*this).id(), state.data_.shape() ) );
             return state.data_.shape();
         }

         std::vector<tensor_type>& contexts()
         {
             auto& state = *((*this).state_);
             return state.contexts_;
         }

         std::vector<tensor_type> contexts()const
{
             auto& state = *((*this).state_);
             return state.contexts_;
         }

         tensor_type& data()
         {
             auto& state = *((*this).state_);
             return state.data_;
         }

         tensor_type data()const
{
             auto& state = *((*this).state_);
             return state.data_;
         }

         tensor_type& gradient()
         {
             auto& state = *((*this).state_);
             return state.gradient_;
         }

         tensor_type gradient()const
{
             auto& state = *((*this).state_);
```

```
            return state.gradient_;
        }

        void reset()
        {
            data().reset();
            gradient().reset();
        }

        /*
void reset_states()
{
if ( stateful_ )
reset();
}
*/

        bool trainable() const noexcept { return trainable_; }
        void trainable( bool t ) { trainable_ = t; }
        /*
bool stateful() const noexcept { return stateful_; }
void stateful( bool s ){ stateful_ = s; }
*/

    };//struct variable

    template< typename T >
    struct is_variable :  std::false_type {};

    template< Tensor Tsor >
    struct is_variable< variable<Tsor> > :  std::true_type {};

    template< class T >
    inline constexpr bool is_variable_v = is_variable<T>::value;

    template< typename T >
    concept Variable = is_variable_v<T>;

    template< Variable Var >
    bool operator == ( Var const& lhs, Var const& rhs ) noexcept
    {
        return lhs.id_ == rhs.id_;
    }

}//namespace ceras

#endif//QVETFVLYKDJJLDBPAMVBUWUGPWXIAIGMXUDVOFGQIHUHVOTBAWEMPJQEWJQIGGSTUCNDHLUYL

```

9.39 /home/feng/workspace/github.repo/ceras/include/xmodel.hpp File Reference

```
#include "./includes.hpp"
#include "./operation.hpp"
#include "./place_holder.hpp"
#include "./session.hpp"
#include "./tensor.hpp"
#include "./utils/better_assert.hpp"
#include "./utils/context_cast.hpp"
#include "./utils/tqdm.hpp"
#include "./utils/list.hpp"
#include "./utils/debug.hpp"
```

Classes

- struct ceras::model< Ex, Ph >

Namespaces

- namespace ceras

9.40 xmodel.hpp

Go to the documentation of this file.

```cpp
#ifndef BPLYFMIFNNWSGMLLEKBJMAJDBRSPHHRAYMOHTWSTCMNMFSLLYNQTTCCAQXKXSLMSLKESHRASL
#define BPLYFMIFNNWSGMLLEKBJMAJDBRSPHHRAYMOHTWSTCMNMFSLLYNQTTCCAQXKXSLMSLKESHRASL

#include "./includes.hpp"
#include "./operation.hpp"
#include "./place_holder.hpp"
#include "./session.hpp"
#include "./tensor.hpp"
#include "./utils/better_assert.hpp"
#include "./utils/context_cast.hpp"
#include "./utils/tqdm.hpp"
#include "./utils/list.hpp"
#include "./utils/debug.hpp"

namespace ceras
{

    template< List Input_List, List Output_List >
    struct model
    {
        Input_List  input_layer_;
        Output_List output_layer_;

        constexpr model( Input_List const& input_layer, Output_List const& output_layer ) : input_layer_{ input_layer }, output_layer_{ output_layer } {}

        auto constexpr input()const { return input_layer_; }

        auto constexpr output()const { return output_layer_; }


        template< Tensor Tsor>
        auto constexpr predict( Tsor const& input_tensor ) const // in case of only one input layer
        {

            //constexpr debug<length( input_layer_ )> = 0;

            better_assert( length( input_layer_ ) == 1, "Expecting the model only have a single input layer, but get ", length( input_layer_ ) );

            auto& s = get_default_session<Tsor>();
            s.bind( car(input_layer_), input_tensor );

            learning_phase = 0; // for different behaviours in normalization and drop-out layers

            if constexpr( length(output_layer_) == 1 )
            {
                // if the model has a single output layer, return a tensor.
                Tsor ans = s.run( car(output_layer_) );
                learning_phase = 1; // restore learning phase
                return ans;
            }
            else
            {
                // if the model has multiple output layer, return a tuple of tensors
                auto ans = map( [&s]<Expression Ex>( Ex const& ex ){ return s.run(ex); }, output_layer_ );
                learning_phase = 1; // restore learning phase
                return ans.as_tuple();
            }
        }

        template< List Tsor_List >
        auto predict( Tsor_List const& input_tensor )const
{
            static_assert( length(input_tensor) == length(input_layer_), "Expecting same number of input layers" );

            if constexpr( length(input_tensor) == 1 )
            {
                // case of single input layer
                return predict( car(input_tensor) );
            }
```

```
            else
            {
                typedef typename std::remove_cv_t<decltype( car(input_tensor) )> tensor_type;
                auto& s = get_default_session<tensor_type>();
                learning_phase = 0; // supress training behaviours in Dropout and BN layers

                map
                (
                     [&s]<List List_PHolder_Tensor>( List_PHolder_Tensor const& list_of_layer_and_tensor ) // invokes on sub-list of input_layer-input-tensor
                     {
                         list_of_layer_and_tensor
                         (
                             [&s]<Place_Holder Ph, Tensor Tsor>(Ph input_layer, Tsor input_tensor) // invokes with an input layer (a place holder) and an input tensor
                             {
                                 s.bind( input_layer, input_tensor );
                             }
                         );
                     },
                     zip( input_layer_, input_tensor ) // pairing input layer and input tensor as a list of list -> [ [input_layer_0, input_tensor_0], [input_layer_1, input_tensor_1], ..., [input_layer_n, input_tensor_n] ]
                 );

                 if constexpr( length(output_layer_) == 1 )
                 {
                     // if the model has a single output layer, return a tensor.
                     tensor_type ans = s.run( car(output_layer_) );
                     learning_phase = 1; // restore learning phase
                     return ans;
                 }
                 else
                 {
                     // if the model has multiple output layer, return a tuple of tensors
                     auto ans = map( [&s]<Expression Ex>( Ex const& ex ){ return s.run(ex); }, output_layer_ );
                     learning_phase = 1; // restore learning phase
                     return ans.as_tuple();
                 }

             }
         }


    }; // struct model


}//namespace ceras

#endif//BPLYFMIFNNWSGMLLEKBJMAJDBRSPHHRAYMOHTWSTCMNMFSLLYNQTTCCAQXKXSLMSLKESHRASL
```

Index