Tensors with Autograd Library Documentation

Adrian Wilhelmi, 268479

May 22, 2024 r.

Contents

1 Introduction

This document provides a comprehensive guide to the Tensors with Autograd Library, covering function descriptions, usage examples, and more.

2 Tensor class member functions

By default, new tensors have gradient tracking turned off, unless they were created by an operation on a tensor, which has gradient tracking turned on. All binary operators are overloaded as element wise operations. Operations on Tensors with different shapes will be broadcasted similairly to PyTorch. Most operations that change tensor's state have thier corresponding in-place substitute. To perform an operation in place, end the function name with _, for example transpose_();

2.1 Tensor creation

Besides the standard constructors Tensor class implements:

```
Tensor(const std::size_t size, const T& fill_val = T(0))
```

Creates 1 dimensional tensor with size (size) and fills it with value (fill_val).

```
Tensor(const TensorSlice& d, Storage<T>&e)
```

Creates a tensor with shape specified by descriptor (d) and shares memory with stroage (e).

```
Tensor(const TensorSlice& d)
```

Creates a tensor with shape specified by descriptor (d).

```
Tensor<T> copy_dims()
```

Returns a tensor with shape copied from *this.

```
void share(Tensor<T>& t)
```

Shares this tensor's storage with tensor (t).

```
template<typename U = bool>
Tensor<U> one_hot(std::size_t num_classes = 0) const;
```

Converts the tensor to a one-hot encoded tensor.
Parameters:

- `num_classes` (std::size_t): Number of classes. If 0, it will be set to max value in tensor + 1.

Returns:

- Tensor<U>: A one-hot encoded tensor.

2.2 Indexing, slicing, shaping operations

```
T& opeartor()(Args... args)
```

Performs classing indexing

```
Tensor<T> operator()(const TensorSlice& d)
```

Returns a tensor that is a view of this tensor's storage using descriptor (d).

```
T& item()
```

Returns tensor's first element. Only relevant on tensors with only one element.

```
Tensor<T> dimslice(const std::size_t n, const std::size_t m);
```

Generalization of row/column to any dimension. row(i) is same as dimslice(0, i) col(i) is same as dimslice(1, i)
Parameters:

- `n` (const std::size_t): dimension from which we choose the subtensor.
- `m` (const std::size_t): position of subtensor.

Returns:

- Tensor<T>: Tensor that is a m-th subtensor of n-th dimension of (this). Assuming the shape of (this) is (a_1, ..., m, ..., a_N), shape of output tensor will be (a_1, ..., a_N).

`Tensor<T> dimslices(const std::size_t n, Args... args);`

Generalization of dimslice to any number of subtensors (LIMITED TO 2 SUBTENSORS FOR NOW).

Parameters:

- `n` (const std::size_t): dimension from which we choose the subtensors.
- `args` (Args...): positions of subtensors.

Returns:

- Tensor<T>: Tensor that is a concatenation of subtensors specialized in (args). Assuming the shape of (this) is (a_1, ..., m, ..., a_N), shape of output tensor will be (a_1, ..., sizeof...(args), ..., a_N).

`Tensor<T> dimslices_range(const std::size_t n, const std::size_t start,`
`↪ const std::size_t end)`

Returns (end) - (start) consecutive subtensors starting from (start)

Parameters:

- `n` (const std::size_t): dimension from which we choose the subtensors.
- `start` (std::size_t): position of first subtensor.
- `end` (std::size_t): position of last subtensor.

Returns:

- Tensor<T>: Tensor that is a concatenation of consecutive subtensors from (start) to (end). Assuming the shape of (this) is (a_1, ..., m, ..., a_N), shape of output tensor will be (a_1, ..., sizeof...(args), ..., a_N).

`Tensor<T> view(Args... args)`

Returns a tensor with the same data and number of elements as input, but with the specified shape.

Parameters:

- `args` (Args...): new shape

Returns:

- Tensor<T>: Reshaped version of (this)

`Tensor<T> reshape(Args... args)`

Same as view, but creates a new tensor.

Parameters:

- `args` (Args...): new shape

Returns:

- Tensor<T>: Reshaped version of (this)

`Tensor<T> argmax()`

Indices of maximum element. targs (Args...): new shape

Returns:

- Tensor<std::size_t>: 1d tensor containing indices of this tensor's maximum element.

`Tensor<T> row(const std::size_t i)`

Returns:

- Tensor<std::size_t>: this->dimslice(0, i)

`Tensor<T> operator[](const std::size_t i)`

same as row(i), C style indexing.

`Tensor<T> col(const std::size_t i)`

Returns:

- Tensor<std::size_t>: this->dimslice(1, i)

`Tensor<T> col(const std::size_t i)`

Returns:

- Tensor<std::size_t>: this->dimslice(1, i)

`Tensor<T> rot180()`

Returns:

- Tensor<std::size_t>: (this) tensor rotated 180 degrees.

`Tensor<T> rot90()`

Returns:

- Tensor<std::size_t>: (this) tensor rotated 90 degrees clockwise.

`Tensor<T> diag()`

Works only on 2d square tensors. **Returns:**

- Tensor<std::size_t>: (this) tensor's diagonal.

```
Tensor<T> transpose(const std::size_t d1 = 0, const std::size_t d2 =
    ↪ 1)
```

Returns:

- Tensor<std::size_t>: (this) tensor with (d1) and (d2) dimensions transposed.

2.3 Utility functions

```
bool shares_data()
```

Returns:

- true: if tensor shares memory with at least 1 other tensor.
- false: else.

```
T sum()
```

Returns:

- sum (T): sum of all tensor's elements.

```
T sum(const std::size_t i)
```

Returns:

- Tensor<T>: 1D tensor containing sums of elements across dimensions speficied by (i).

```
T mean()
```

Returns:

- mean (T): mean of all tensor's elements.

```
T max()
```

Returns:

- max (T): tensor's maximum element.

```
T min()
```

Returns:

- min (T): tensor's minimum element.

```
T median()
```

Returns:

- median (T): median of all tensor's elements.

`T var()`

Returns:

- var (T): variance of all tensor's elements.

`T std()`

Returns:

- var (T): standard deviation of all tensor's elements.

`T lp_norm(const float p)`

Returns:

- norm (T): Lp norm of all tensor's elements.

`void sort()`

Sorts tensor's elements ascending.

`T* data()`

Returns:

- data (T*): pointer to the first tensor's element.

`std::size_t order()`

Returns:

- order (std::size_t): (this) tensor's order.

`std::size_t extent(std::size_t i)`

Returns:

- extent (std::size_t): shape of (this) tensor at dimension (i).

`std::size_t size()`

Returns:

- size (std::size_t): number of (this) tensor's elements.

`TensorSlice descriptor()`

Returns:

- desc (TensorSlice): Object representing (this) tensors shape.

```
Storage<T>& storage()
```

Returns:

- Storage<T>: Object representing (this) tensors elements.

```
Storage<T>& storage()
```

Returns:

- Storage<T>: Object representing (this) tensors elements.

```
template<typename U>
Tensor<U> convert()
```

Returns:

- Tensor<U>: (this) tensor with type T converted to type U.

```
void swap(Tensor<T>& other);
```

Swaps the contents of the current tensor with another tensor.

Parameters:

- `other` (Tensor<T>&): Tensor to swap with.

Throws:

- std::runtime_error: if tensors have different shapes.

```
void shuffle_(Tensor<T>& other, const std::size_t dim = 0);
```

Shuffles the tensor along a specified dimension and also does it the same way to tensor (other)

Parameters:

- `other` (Tensor<T>&): Tensor to swap with during shuffle.
- `dim` (std::size_t): Dimension to shuffle along.

Throws:

- std::runtime_error: if tensors have different shapes along the specified dimension.

```
void shuffle_(const std::size_t dim = 0);
```

Shuffles the tensor along a specified dimension.

Parameters:

- `dim` (std::size_t): Dimension to shuffle along.

```
Tensor<T> cumsum();
```

Computes the cumulative sum of the tensor elements.
Returns:

- Tensor<T>: A tensor with cumulative sums of the original tensor's elements.

```
Tensor<T> clip(const T low, const T high);
```

Clips the tensor's elements to be within the specified range and returns a new tensor.
Parameters:

- `low` (T): Lower bound.
- `high` (T): Upper bound.

Returns:

- Tensor<T>: A new tensor with elements clipped to the specified range.

Throws:

- std::runtime_error: if `low` is greater than `high`.

2.4 Math operations

```
Tensor<T> pow(Tensor<T>& exps);
```

Raises each element of the tensor to the power of the corresponding element in another tensor.
Parameters:

- `exps` (Tensor<T>&): Tensor of exponents.

Returns:

- Tensor<T>: A new tensor with elements raised to the corresponding powers.

```
Tensor<T> pow(const T exp) const;
```

Raises each element of the tensor to a specified power.
Parameters:

- `exp` (T): Exponent.

Returns:

- Tensor<T>: A new tensor with each element raised to the specified power.

```
Tensor<T> sqrt() const;
```

Computes the square root of each element in the tensor.
Returns:
- Tensor<T>: A new tensor with the square root of each element.

```
Tensor<T> log();
```

Computes the natural logarithm of each element in the tensor.
Returns:
- Tensor<T>: A new tensor with the natural logarithm of each element.

```
Tensor<T> exp();
```

Computes the exponential of each element in the tensor.
Returns:
- Tensor<T>: A new tensor with the exponential of each element.

```
Tensor<T> relu();
```

Applies the ReLU activation function element-wise.
Returns:
- Tensor<T>: A new tensor with ReLU applied to each element.

```
Tensor<T> tanh();
```

Applies the tanh activation function element-wise.
Returns:
- Tensor<T>: A new tensor with tanh applied to each element.

```
Tensor<T> sigmoid();
```

Applies the sigmoid activation function element-wise.
Returns:
- Tensor<T>: A new tensor with sigmoid applied to each element.

```
Tensor<T> softmax();
```

Applies the softmax function to the tensor.
Returns:
- Tensor<T>: A new tensor with softmax applied.

2.5 Autograd related functions

```
bool requires_grad() const;
```

Checks if gradient tracking is enabled for the tensor.
Returns:
- bool: True if gradient tracking is enabled, false otherwise.

```
void enable_grad();
```

Enables gradient tracking for the tensor.

```
Tensor<T>& grad();
```

Returns the gradients of the tensor.
Returns:
- Tensor<T>&: The gradients of the tensor.

Throws:

- std::runtime_error: if gradients are not enabled.

```
Tensor<T> grad(const TensorSlice& d);
```

Returns the gradients of the tensor for the specified slice.
Parameters:
- `d` (const TensorSlice&): Slice descriptor.

Returns:

- Tensor<T>: The gradients of the tensor for the specified slice.

Throws:

- std::runtime_error: if gradients are not enabled.

```
void zero_grad();
```

Sets the gradients of the tensor to zero.
Throws:
- std::runtime_error: if gradients are not enabled.

```
void backward();
```

Performs backpropagation through the computation graph starting from the current tensor and initializes gradient to 1 if not already set.
Throws:
- std::runtime_error: if gradients are not enabled.

3 tensor namespace

`Tensor<T> Tensor::ones(Exts... exts)`

Creates a tensor of dimensions specified in input filled with ones.
Parameters:
- `exts...` (std::size_t): Tensor's shape.

Returns:

- Tensor<T>: A tensor of dimensions (exts...) filled with ones.

`Tensor<T> Tensor::zeros(Exts... exts)`

Creates a tensor of dimensions specified in input filled with zeros.
Parameters:
- `exts...` (std::size_t): Tensor's shape.

Returns:

- Tensor<T>: A tensor of dimensions (exts...) filled with zeros.

`Tensor<T> Tensor::zeros(Tensor<T>& t)`

Creates a tensor with shape copied from tensor from input and fills it with zeros.
Parameters:
- `exts...` (std::size_t): Tensor's shape.

Returns:

- Tensor<T>: A tensor of dimensions (exts...) filled with zeros.

`Tensor<T> Tensor::ones(Tensor<T>& t)`

Creates a tensor with shape copied from tensor from input and fills it with ones.
Parameters:
- `exts...` (std::size_t): Tensor's shape.

Returns:

- Tensor<T>: A tensor of dimensions (exts...) filled with ones.

```
template<typename T, typename U>
bool same_storage(const Tensor<T>& t1, const Tensor<U>& t2);
```

Checks if two tensors share the same underlying storage.

Parameters:

- `t1` (const Tensor<T>&): The first tensor.
- `t2` (const Tensor<U>&): The second tensor.

Returns:

- bool: True if tensors share the same storage, false otherwise.

```
template<typename T, std::size_t N>
Tensor<T> from_list(const TensorInitializer<T, N>& init, bool req_grad
    ↪ = false);
```

Creates a tensor from a nested list initializer.

Parameters:

- `init` (const TensorInitializer<T, N>&): Nested list initializer.
- `req_grad` (bool): If true, enables gradient tracking for the tensor.

Returns:

- Tensor<T>: The created tensor.

```
template<typename T>
Tensor<T> from_image(const std::string& filepath);
```

Loads an image from a file and creates a tensor from it.

Parameters:

- `filepath` (const std::string&): Path to the image file.

Returns:

- Tensor<T>: Tensor representing the image.

Throws:

- std::runtime_error: if the image file cannot be opened or if the type is not float or int.

```
template<typename T>
Tensor<T> from_video(const std::string& filepath);
```

Loads a video from a file and creates a tensor from it.
Parameters:

- `filepath` (const std::string&): Path to the video file.

Returns:

- Tensor<T>: Tensor representing the video.

Throws:

- std::runtime_error: if the video file cannot be opened or if the type is not float or int.

```
template<typename T, typename... Exts>
Tensor<T> zeros(Exts... exts);
```

Creates a tensor filled with zeros.
Parameters:

- `exts` (Exts...): Dimensions of the tensor.

Returns:

- Tensor<T>: Tensor filled with zeros.

```
template<typename T, typename... Exts>
Tensor<T> ones(Exts... exts);
```

Creates a tensor filled with ones.
Parameters:

- `exts` (Exts...): Dimensions of the tensor.

Returns:

- Tensor<T>: Tensor filled with ones.

```
template<typename T, typename... Exts>
Tensor<T> random_normal(const T mean, const T stddev, const Exts...
    ↪ exts);
```

Creates a tensor with elements drawn from a normal distribution.
Parameters:

- `mean` (const T): Mean of the distribution.
- `stddev` (const T): Standard deviation of the distribution.
- `exts` (Exts...): Dimensions of the tensor.

Returns:

- Tensor<T>: Tensor with normally distributed elements.

```
template<typename T>
Tensor<T> random_normal(const T mean, const T stddev, const Tensor<T>&
    ↪ t);
```

Creates a tensor with elements drawn from a normal distribution, with the same dimensions as another tensor.

Parameters:

- `mean` (const T): Mean of the distribution.
- `stddev` (const T): Standard deviation of the distribution.
- `t` (const Tensor<T>&): Tensor to copy dimensions from.

Returns:

- Tensor<T>: Tensor with normally distributed elements.

```
template<typename T, typename... Exts>
Tensor<T> random_bernoulli(const double p, Exts... exts);
```

Creates a tensor with elements drawn from a Bernoulli distribution.

Parameters:

- `p` (const double): Probability of success.
- `exts` (Exts...): Dimensions of the tensor.

Returns:

- Tensor<T>: Tensor with Bernoulli distributed elements.

```
template<typename T>
Tensor<T> random_bernoulli(const double p, const Tensor<T>& t);
```

Creates a tensor with elements drawn from a Bernoulli distribution, with the same dimensions as another tensor.

Parameters:

- `p` (const double): Probability of success.
- `t` (const Tensor<T>&): Tensor to copy dimensions from.

Returns:

- Tensor<T>: Tensor with Bernoulli distributed elements.

```
template<typename T, typename... Exts>
Tensor<T> random_uniform(const T min, const T max, const Exts... exts)
    ↪ ;
```

Creates a tensor with elements drawn from a uniform distribution.
Parameters:

- `min` (const T): Minimum value of the distribution.
- `max` (const T): Maximum value of the distribution.
- `exts` (Exts...): Dimensions of the tensor.

Returns:

- Tensor<T>: Tensor with uniformly distributed elements.

```
template<typename T>
Tensor<T> random_uniform(const T min, const T max, const Tensor<T>& t)
    ↪ ;
```

Creates a tensor with elements drawn from a uniform distribution, with the same dimensions as another tensor.
Parameters:

- `min` (const T): Minimum value of the distribution.
- `max` (const T): Maximum value of the distribution.
- `t` (const Tensor<T>&): Tensor to copy dimensions from.

Returns:

- Tensor<T>: Tensor with uniformly distributed elements.

```
template<typename T = int, typename... Exts>
Tensor<T> randint(const int min, const int max, const Exts... exts);
```

Creates a tensor with elements drawn from a uniform integer distribution.
Parameters:

- `min` (const int): Minimum value of the distribution.
- `max` (const int): Maximum value of the distribution.
- `exts` (Exts...): Dimensions of the tensor.

Returns:

- Tensor<T>: Tensor with uniformly distributed integer elements.

```
template<typename T = int>
Tensor<T> randint(const int min, const int max, const Tensor<T>& t);
```

Creates a tensor with elements drawn from a uniform integer distribution, with the same dimensions as another tensor.

Parameters:

- `min` (const int): Minimum value of the distribution.
- `max` (const int): Maximum value of the distribution.
- `t` (const Tensor<T>&): Tensor to copy dimensions from.

Returns:

- Tensor<T>: Tensor with uniformly distributed integer elements.

```
template<typename U, typename T = std::size_t>
Tensor<T> random_multinomial(const Tensor<U>& probs, std::size_t
    ↪ num_samples, bool replacement = true);
```

Draws samples from a multinomial distribution.

Parameters:

- `probs` (const Tensor<U>&): Tensor of probabilities.
- `num_samples` (std::size_t): Number of samples to draw.
- `replacement` (bool): Whether to sample with replacement.

Returns:

- Tensor<T>: Tensor of sampled indices.

Throws:

- std::runtime_error: if probabilities do not sum to 1.

```
template<typename T>
Tensor<T> eye(const std::size_t n)
```

Creates an identity matrix.

Parameters:

- `n` (std::size_t): Size of the identity matrix.

Returns:

- Tensor<T>: Identity matrix of size `n` by `n`.

```
template<typename T>
Tensor<T> arange(const T start, const T end, const T step = 1);
```

Creates a tensor with values ranging from `start` to `end` with a given `step`.

Parameters:

- `start` (const T): Starting value of the range.
- `end` (const T): Ending value of the range (exclusive).
- `step` (const T): Step size between values.

Returns:

- Tensor<T>: Tensor with values in the specified range.

Throws:

- std::invalid_argument: if `step` is zero.

```
template<typename T>
void to_image(const Tensor<T>& tensor, const std::string& filepath);
```

Saves a tensor as an image to a file.

Parameters:

- `tensor` (const Tensor<T>&): Tensor representing the image.
- `filepath` (const std::string&): Path to save the image.

Throws:

- std::runtime_error: if the tensor does not represent an image.

```
template<typename T>
void to_video(const Tensor<T>& tensor, const std::string& filepath);
```

Saves a tensor as a video to a file.

Parameters:

- `tensor` (const Tensor<T>&): Tensor representing the video.
- `filepath` (const std::string&): Path to save the video.

Throws:

- std::runtime_error: if the tensor does not represent a video.

```
template<typename T, typename U = bool>
Tensor<U> one_hot(const Tensor<T>& t, std::size_t num_classes = 0);
```

Converts a tensor of class indices to a one-hot encoded tensor.

Parameters:

- `t` (const Tensor<T>&): Tensor of class indices.
- `num_classes` (std::size_t): Number of classes (optional).

Returns:

- Tensor<U>: One-hot encoded tensor.

```
template<typename T>
bool nearly_equal(const Tensor<T>& t1, const Tensor<T>& t2);
```

Checks if two tensors are nearly equal within a tolerance.

Parameters:

- `t1` (const Tensor<T>&): The first tensor.
- `t2` (const Tensor<T>&): The second tensor.

Returns:

- bool: True if tensors are nearly equal, false otherwise.

```
template<typename T>
Tensor<T> concat(Tensor<T>& t1, Tensor<T>& t2, std::size_t dim);
```

Concatenates two tensors along a specified dimension.

Parameters:

- `t1` (Tensor<T>&): The first tensor.
- `t2` (Tensor<T>&): The second tensor.
- `dim` (std::size_t): Dimension to concatenate along.

Returns:

- Tensor<T>: The concatenated tensor.

Throws:

- std::runtime_error: if tensors have different orders or mismatched extents (except along the specified dimension).

```
template<typename T>
Tensor<T> concat(const Tensor<T>& t1, const Tensor<T>& t2, std::size_t
    ↪ dim);
```

Concatenates two tensors along a specified dimension (constant version).

Parameters:

- `t1` (const Tensor<T>&): The first tensor.
- `t2` (const Tensor<T>&): The second tensor.
- `dim` (std::size_t): Dimension to concatenate along.

Returns:

- Tensor<T>: The concatenated tensor.

Throws:

- std::runtime_error: if tensors have different orders or mismatched extents (except along the specified dimension).

```
template<typename T>
Tensor<T> mean(Tensor<T>& t);
```

Computes the mean of the tensor elements.

Parameters:

- `t` (Tensor<T>&): The input tensor.

Returns:

- Tensor<T>: Tensor with the mean value of the elements.

```
template<typename T>
T dot(const Tensor<T>& t1, const Tensor<T>& t2);
```

Computes the dot product of two tensors.

Parameters:

- `t1` (const Tensor<T>&): The first tensor.
- `t2` (const Tensor<T>&): The second tensor.

Returns:

- T: The dot product of the two tensors.

Throws:

- std::runtime_error: if the tensors have different sizes.

```
template<typename T>
Tensor<T> matmul(Tensor<T>& t1, Tensor<T>& t2);
```

Performs matrix multiplication of two 2D tensors.
Parameters:

- `t1` (Tensor<T>&): The first tensor (matrix).
- `t2` (Tensor<T>&): The second tensor (matrix).

Returns:

- Tensor<T>: The result of the matrix multiplication.

Throws:

- std::runtime_error: if tensors are not 2D or if the dimensions do not match for matrix multiplication.

```
template<typename T>
Tensor<T> matmul(const Tensor<T>& t1, const Tensor<T>& t2);
```

Performs matrix multiplication of two 2D tensors (constant version).
Parameters:

- `t1` (const Tensor<T>&): The first tensor (matrix).
- `t2` (const Tensor<T>&): The second tensor (matrix).

Returns:

- Tensor<T>: The result of the matrix multiplication.

Throws:

- std::runtime_error: if tensors are not 2D or if the dimensions do not match for matrix multiplication.

```
template<typename T>
Tensor<T> matmul_classic(Tensor<T>& t1, Tensor<T>& t2);
```

Performs classic matrix multiplication of two 2D tensors using a nested loop approach.
Parameters:

- `t1` (Tensor<T>&): The first tensor (matrix).
- `t2` (Tensor<T>&): The second tensor (matrix).

Returns:

- Tensor<T>: The result of the matrix multiplication.

Throws:

- std::runtime_error: if tensors are not 2D or if the dimensions do not match for matrix multiplication.

```
template<typename T>
Tensor<T> matmul_classic(const Tensor<T>& t1, const Tensor<T>& t2);
```

Performs classic matrix multiplication of two 2D tensors using a nested loop approach (constant version).

Parameters:

- `t1` (const Tensor<T>&): The first tensor (matrix).
- `t2` (const Tensor<T>&): The second tensor (matrix).

Returns:

- Tensor<T>: The result of the matrix multiplication.

Throws:

- std::runtime_error: if tensors are not 2D or if the dimensions do not match for matrix multiplication.

```
template<typename T>
Tensor<T> conv2d(Tensor<T>& input, Tensor<T>& kernel, const std::
    ↪ size_t stride = 1, const std::size_t padding = 1);
```

Performs a 2D convolution operation on the input tensor with the given kernel.

Parameters:

- `input` (Tensor<T>&): The input tensor.
- `kernel` (Tensor<T>&): The convolution kernel.
- `stride` (std::size_t): Stride of the convolution. Default is 1.
- `padding` (std::size_t): Padding added to the input tensor. Default is 1.

Returns:

- Tensor<T>: The result of the convolution operation.

Throws:

- std::runtime_error: if the input and kernel do not have the same number of channels.

```
template<typename T>
Tensor<T> conv2d(const Tensor<T>& input, const Tensor<T>& kernel,
    ↪ const std::size_t stride = 1, const std::size_t padding = 1);
```

Performs a 2D convolution operation on the input tensor with the given kernel (constant version).

Parameters:

- `input` (const Tensor<T>&): The input tensor.
- `kernel` (const Tensor<T>&): The convolution kernel.
- `stride` (std::size_t): Stride of the convolution. Default is 1.
- `padding` (std::size_t): Padding added to the input tensor. Default is 1.

Returns:

- Tensor<T>: The result of the convolution operation.

Throws:

- std::runtime_error: if the input and kernel do not have the same number of channels.

```
template<typename T>
Tensor<T> max_pooling(Tensor<T>& input, std::size_t kernel_size, std::
    ↪ size_t stride);
```

Performs max pooling on the input tensor.

Parameters:

- `input` (Tensor<T>&): The input tensor.
- `kernel_size` (std::size_t): Size of the pooling kernel.
- `stride` (std::size_t): Stride of the pooling operation.

Returns:

- Tensor<T>: The result of the max pooling operation.

Throws:

- std::invalid_argument: if the input tensor is not 3D.

```
template<typename T>
Tensor<T> max_pooling(const Tensor<T>& input, std::size_t kernel_size,
    ↪ std::size_t stride = 1);
```

Performs max pooling on the input tensor (constant version).
Parameters:

- `input` (const Tensor<T>&): The input tensor.
- `kernel_size` (std::size_t): Size of the pooling kernel.
- `stride` (std::size_t): Stride of the pooling operation. Default is 1.

Returns:

- Tensor<T>: The result of the max pooling operation.

Throws:

- std::invalid_argument: if the input tensor is not 3D.

```
template<typename T>
Tensor<T> avg_pooling(Tensor<T>& input, std::size_t kernel_size, std::
    ↪ size_t stride);
```

Performs average pooling on the input tensor.
Parameters:

- `input` (Tensor<T>&): The input tensor.
- `kernel_size` (std::size_t): Size of the pooling kernel.
- `stride` (std::size_t): Stride of the pooling operation.

Returns:

- Tensor<T>: The result of the average pooling operation.

Throws:

- std::invalid_argument: if the input tensor is not 3D.

```
template<typename T>
Tensor<T> avg_pooling(const Tensor<T>& input, std::size_t kernel_size,
    ↪ std::size_t stride = 1);
```

Performs average pooling on the input tensor (constant version).
Parameters:

- `input` (const Tensor<T>&): The input tensor.
- `kernel_size` (std::size_t): Size of the pooling kernel.
- `stride` (std::size_t): Stride of the pooling operation. Default is 1.

Returns:

- Tensor<T>: The result of the average pooling operation.

Throws:

- std::invalid_argument: if the input tensor is not 3D.

```
template<typename T>
Tensor<T> cross_entropy(Tensor<T>& logits, Tensor<T>& targets);
```

Computes the cross-entropy loss between the logits and targets.
Parameters:

- `logits` (Tensor<T>&): Tensor of logits.
- `targets` (Tensor<T>&): Tensor of targets.

Returns:

- Tensor<T>: The cross-entropy loss.

Throws:

- std::invalid_argument: if the orders or extents of the logits and targets do not match.

```
template<typename T>
Tensor<T> cross_entropy(const Tensor<T>& logits, const Tensor<T>&
    ↪ targets);
```

Computes the cross-entropy loss between the logits and targets (constant version).
Parameters:

- `logits` (const Tensor<T>&): Tensor of logits.
- `targets` (const Tensor<T>&): Tensor of targets.

Returns:

- Tensor<T>: The cross-entropy loss.

Throws:

- std::invalid_argument: if the orders or extents of the logits and targets do not match.