Proj2

Release 1.0

Jodok Vieli, Puck van Gerwen, Felix Hoppe

May 25, 2021

CONTENTS

0.1 Project2 Documentation

Project2 is a minimal tensor library for deep learning using CPUs. The current version can:

- Build networks combining fully connected layers, Tanh and RELU
- Run forward and backward passes
- Optimize parameters with SGD and Adam optimizers for MSE

The deep learning functionality is in the `nn` module and the `trainer` can be used to facilitate training models.

0.2 Project2

0.2.1 nn package

Submodules

nn.activation module

class nn.activation.**Activation**
Bases: *nn.module.Module*

backward(*dy*)
Compute gradients of input.

Parameters **dy** (*torch.tensor*) – Backpropagated gradient from the next layer.

Returns Gradient

Return type grad (torch.tensor)

forward(*x*)
Compute the activation.

Parameters **x** (*torch.tensor*) – Input tensor.

class nn.activation.**ReLU**
Bases: *nn.activation.Activation*

ReLU activation function

forward(*x*)
Compute the activation.

Parameters **x** (*torch.tensor*) – Input tensor.

Returns Output tensor.

Return type out (torch.tensor)

class nn.activation.**Tanh**
Bases: *nn.activation.Activation*

Tanh activation function

forward(*x*)
Compute the activation.

Parameters **x** (*torch.tensor*) – Input tensor.

Returns Output tensor.

Return type out (torch.tensor)

nn.functional module

The functional.py contains the concrete implementations of specific functionals.

nn.functional.**d_mse**(*x*, *y*)
Compute the gradient of the mean squared error.

Parameters

- **x** (*torch.tensor*) – Input tensor.
- **y** (*torch.tensor*) – Target tensor.

Returns Gradient of mean squared error.

Return type d_mse (float)

nn.functional.**d_relu**(*x*)
Compute gradient of ReLU(x)

Parameters **x** (*torch.tensor*) – Input tensor

Returns Output tensor

Return type out (torch.tensor)

nn.functional.**d_tanh**(*x*)
Compute gradient of tanh(x)

Parameters **x** (*torch.tensor*) – Input tensor

Returns Output tensor

Return type out (torch.tensor)

nn.functional.**mse**(*x*, *y*)
Compute the mean squared error.

Parameters

- **x** (*torch.tensor*) – Input tensor.
- **y** (*torch.tensor*) – Target tensor.

Returns Mean squared error.

Return type ms_error (torch.tensor)

nn.functional.**relu**(*x*)
Compute ReLU(x)

Parameters **x** (*torch.tensor*) – Input tensor

Returns Output tensor

Return type out (torch.tensor)

nn.functional.**tanh**(*x*)
Compute tanh(x).

Parameters **x** (*torch.tensor*) – Input tensor

Returns Output tensor

Return type out (torch.tensor)

nn.linear module

class nn.linear.**Layer**
Bases: *nn.module.Module*

param()
Return the params of the Layer.

update_param(**args*, ***kwargs*)
Update the params of the Layer based on the cached gradients

class nn.linear.**Linear**(*dim_in*, *dim_out*)
Bases: *nn.linear.Layer*

backward(*dy*)
Compute gradients of input and parameters.

Parameters **dy** (*torch.tensor*) – Backpropagated gradient from the next layer.

Returns Gradient.

Return type output (torch.tensor)

forward(*x*)
Calculate output.

Parameters **x** (*torch.tensor*) – Input tensor of size (batch_size, input_dim)

Returns Output tensor of size (batch_size, output_dim)

Return type output (torch.tensor)

param()
Get parameters of the linear layer from the cache.

Returns weight and bias of linear layer.

Return type w, b (torch.tensor)

nn.linear.**empty**(**size*, ***, *out=None*, *dtype=None*, *layout=torch.strided*, *device=None*, *requires_grad=False*, *pin_memory=False*) → Tensor
Returns a tensor filled with uninitialized data. The shape of the tensor is defined by the variable argument `size`.

Parameters **size** (*int...*) – a sequence of integers defining the shape of the output tensor. Can be a variable number of arguments or a collection like a list or tuple.

Keyword Arguments

- **out** (*Tensor, optional*) – the output tensor.
- **dtype** (`torch.dtype`, optional) – the desired data type of returned tensor. Default: if `None`, uses a global default (see `torch.set_default_tensor_type()`).
- **layout** (`torch.layout`, optional) – the desired layout of returned Tensor. Default: `torch.strided`.
- **device** (`torch.device`, optional) – the desired device of returned tensor. Default: if `None`, uses the current device for the default tensor type (see `torch.set_default_tensor_type()`). `device` will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
- **requires_grad** (*bool, optional*) – If autograd should record operations on the returned tensor. Default: `False`.
- **pin_memory** (*bool, optional*) – If set, returned tensor would be allocated in the pinned memory. Works only for CPU tensors. Default: `False`.

- **memory_format** (torch.memory_format, optional) – the desired memory format of returned Tensor. Default: torch.contiguous_format.

Example:

```
>>> torch.empty(2, 3)
tensor(1.00000e-08 *
       [[ 6.3984,  0.0000,  0.0000],
        [ 0.0000,  0.0000,  0.0000]])
```

nn.loss module

class nn.loss.**Loss**

Bases: *nn.module.Module*

The Loss Module is used to implement a node in the network that computes the loss. For the computation of any function the respective functional from functional.py should be used.

backward()

Backward pass. The backward method can be implemented in the generic Loss class as it should be the same for all Loss Modules.

Returns Backpropagated gradient from the next layer.

Return type dy (torch.tensor)

forward(*x*, *y*)

Compute the loss. :param x: Input tensor. :type x: torch.tensor :param y: Target tensor. :type y: torch.tensor

class nn.loss.**MSELoss**

Bases: *nn.loss.Loss*

forward(*x*, *y*)

Compute the mean squared error.

Parameters

- **x** (*torch.tensor*) – Input tensor.
- **y** (*torch.tensor*) – Target tensor.

Returns Mean squared error.

Return type output (torch.tensor)

nn.module module

class nn.module.**Module**

Bases: object

backward(**args*, ***kwargs*)

forward(**args*, ***kwargs*)

nn.sequential module

class nn.sequential.**Sequential**(*modules*, *loss_fn*)
Bases: *nn.module.Module*

backward()
Perform backward pass.

forward(*x*)
Perform forward pass.

Parameters **x** (*torch.tensor*) – Input tensor

Returns Output tensor

Return type out (torch.tensor)

loss(*x*, *y*)
Compute loss between two tensors.

Parameters

- **x** (*torch.tensor*) – Input tensor
- **y** (*torch.tensor*) – Target tensor

print()
Print model architecture.

test_step(*x*, *y*)

training_step(*x*, *y*)
Training step.

Parameters

- **x** (*torch.tensor*) – Input tensor
- **y** (*torch.tensor*) – Target tensor

Returns computed loss

Return type loss (torch.tensor)

update_params(*optim*, *lr*)

Update the parameters of the network iteratively according to the cached gradients at each module.

Parameters

- **optim** (*string*) – The optimizer to use. options are 'adam' or 'sgd'
- **lr** (*float*) – Learning rate

validation_step(*x*, *y*)
Validation step.

Parameters

- **x** (*torch.tensor*) – Input tensor
- **y** (*torch.tensor*) – Target tensor

Returns computed loss

Return type loss (torch.tensor)

Module contents

0.2.2 trainer module

class trainer.**Trainer**(*nb_epochs*)

Bases: object

fit(*model*, *x_train*, *y_train*, *x_val*, *y_val*, *batch_size=32*, *lr=0.01*, *optim='sgd'*, *verbose=True*, *print_every=32*)

Train the model on the specified data and print the training and validation loss and accuracy. :param model: Module. Model to train :param dl_train: DataLoader. DataLoader containing the training data :param dl_val: DataLoader. DataLoader containting the validation data :param verbose: bool. Whether or not to output training information

Example

Use the trainer to fit a new nn model.

```
>>> from trainer import Trainer
>>> from torch import empty
>>> from nn.sequential import Sequential
>>> ... # Read data into x_train, y_train, x_test, y_test
>>> LinNet = LinNet = Sequential((Linear(2, 1), MSELoss())
>>> trainer = Trainer(nb_epochs=25)
>>> loss_train, loss_val = trainer.fit(LinNet, x_train, y_train, x_test, y_
↪test, batch_size=32, lr=0.1, print_every=10, optim='sgd')
```

test(*model*, *x_test*, *y_test*, *batch_size=32*, *test_verbose=True*)

Test the model on the specified data :param model: Module. Model to train :param dl_test: DataLoader. DataLoader containting the test data :param test_verbose: bool. Whether the test result should be printed

Example

Use the trainer to test an existing nn model.

```
>>> from trainer import Trainer
>>> from torch import empty
>>> ... # Train model LinNet
>>> ... # Read data into x_train, y_train, x_test, y_test
>>> trainer = Trainer(nb_epochs=25)
>>> loss_test = t.test(LinNet, x_test, y_test, batch_size=32, test_
↪verbose=True)
loss_test=0.17
```

PYTHON MODULE INDEX

n

t

INDEX

U

V