Project 2
Release 1.0

Jodok Vieli[1], Puck van Gerwen[2], Felix Hoppe[3]

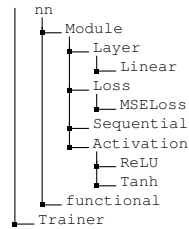**May 26, 2021**

[1]SCIPER: 336284
[2]SCIPER: 283530
[3]SCIPER: 276062

1.0 Project2

Project2 is a minimal tensor library for deep learning using CPUs. Our 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. The library has the following structure:

```
nn
  Module
    Layer
      Linear
    Loss
      MSELoss
    Sequential
    Activation
      ReLU
      Tanh
  functional
Trainer
```

A short example of how to use `Trainer` can be found in section 1.1.1. A test file `test.py` uses the library to train a fully connected network with 3 hidden layers on a toy 2D dataset of 1000 randomly generated points where points inside a circle of radius $1/\sqrt{2\pi}$ are labelled 1, and all others labelled 0. Training for 100 epochs using SGD yields an error rate of 4.81% on average over 10 folds.

1.1 Python API

1.1.0 nn package

1.1.0.0 Submodules

1.1.0.1 nn.activation module

class nn.activation.**Activation**
> Bases: *nn.module.Module*
>
> Class to compute activation functions.
>
> **backward**(*dy*)
>> Compute gradients of input.
>>
>>> **Parameters** **dy** (*torch.tensor*) – Backpropagated gradient from the next layer.
>>>
>>> **Returns** Gradient
>>>
>>> **Return type** torch.tensor
>
> **forward**(*x*)
>> Compute the activation.
>>
>>> **Parameters** **x** (*torch.tensor*) – Input tensor.

class nn.activation.**ReLU**
> Bases: *nn.activation.Activation*
>
> **forward**(*x*)
>> Compute ReLU(x)
>>
>>> **Parameters** **x** (*torch.tensor*) – Input tensor.
>>>
>>> **Returns** Computed ReLU.
>>>
>>> **Return type** torch.tensor

class nn.activation.**Tanh**
> Bases: *nn.activation.Activation*
>
> **forward**(*x*)
>> Compute tanh(x)
>>
>>> **Parameters** **x** (*torch.tensor*) – Input tensor.
>>>
>>> **Returns** Computed tanh(x).
>>>
>>> **Return type** torch.tensor

1.1.0.2 nn.functional module

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** float

nn.functional.**d_relu**(*x*)
> Compute gradient of ReLU(x)
>
>> **Parameters** **x** (*torch.tensor*) – Input tensor
>>
>> **Returns** Output tensor
>>
>> **Return type** torch.tensor

nn.functional.**d_tanh**(*x*)
> Compute gradient of tanh(x)
>
>> **Parameters** **x** (*torch.tensor*) – Input tensor
>>
>> **Returns** Output tensor
>>
>> **Return type** 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** torch.tensor

nn.functional.**relu**(*x*)
> Compute ReLU(x)
>
>> **Parameters** **x** (*torch.tensor*) – Input tensor
>>
>> **Returns** Output tensor
>>
>> **Return type** torch.tensor

nn.functional.**tanh**(*x*)
> Compute tanh(x).
>
>> **Parameters** **x** (*torch.tensor*) – Input tensor
>>
>> **Returns** Output tensor
>>
>> **Return type** torch.tensor

1.1.0.3 nn.linear module

class nn.linear.**Layer**
> Bases: *nn.module.Module*
>
> Layer implements layers that can be used in a network architecture.
>
> **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** torch.tensor
>
> **forward**(*x*)
>> Calculate output of Linear layer.
>>
>>> **Parameters** **x** (*torch.tensor*) – Input tensor of size (batch_size, input_dim)
>>>
>>> **Returns** Output tensor of size (batch_size, output_dim)
>>>
>>> **Return type** torch.tensor
>
> **param**()
>> Get parameters of the linear layer from the cache.
>>
>>> **Returns** weight and bias of linear layer.
>>>
>>> **Return type** torch.tensor, torch.tensor

1.1.0.4 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.
>>
>>> **Returns** Backpropagated gradient from the next layer.
>>>
>>> **Return type** torch.tensor
>
> **forward**(*x, y*)

Compute the loss.

Parameters

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

1.1.0.5 nn.module module

class nn.module.**Module**
Bases: `object`

Base Module with core functionality

backward (**args*, ***kwargs*)
Compute backward pass

forward (**args*, ***kwargs*)
Compute forward pass

1.1.0.6 nn.sequential module

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

Sequential allows multiple layers to be combined in a network architecture.

backward ()
Perform backward pass.

forward (*x*)
Perform forward pass.

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

Returns Output tensor

Return type torch.tensor

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

Parameters

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

Returns Loss

Return type torch.tensor

print ()
str: Print model architecture.

test_step (*x*, *y*)
Test step. Wrapper for validation_step.

Parameters

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

Returns Loss

Return type torch.tensor

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

Parameters

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

Returns Loss

Return type 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. op-

tions 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 Loss

Return type torch.tensor

1.1.1 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*, *op-tim='sgd'*, *verbose=True*, *print_every=32*)
Train the model on the specified data and print the training and validation loss and accuracy.

Parameters

- **model** (`nn.Module`) – Model to train
- **x_train** (`torch.tensor`) – Training data
- **y_train** (`torch.tensor`) – Training labels
- **x_val** (`torch.tensor`) – Validation data
- **y_val** (`torch.tensor`) – Validation labels
- **batch_size** (`int`) – Batch sizes for training and validation
- **lr** (`float`) – Learning rate for optimization (Default is 0.01)
- **optim** (`str`) – Optimizer (options are 'sgd' or 'adam'. Default is 'sgd')
- **verbose** (`bool`) – Whether or not to output training information (Default is True)
- **print_every** (`str`) – How often to print progress (Default is every 32 steps)

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 = 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.

Parameters

- **model** (`nn.Module`) – Model to train
- **x_test** (`torch.tensor`) – Test data
- **y_test** (`torch.tensor`) – Test labels
- **batch_size** (`int`) – Batch size for testing
- **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
```