PyTorch Introduction
Training a network

David Rotermund & Udo Ernst

Recommendation

Open Book: Dive into Deep Learning

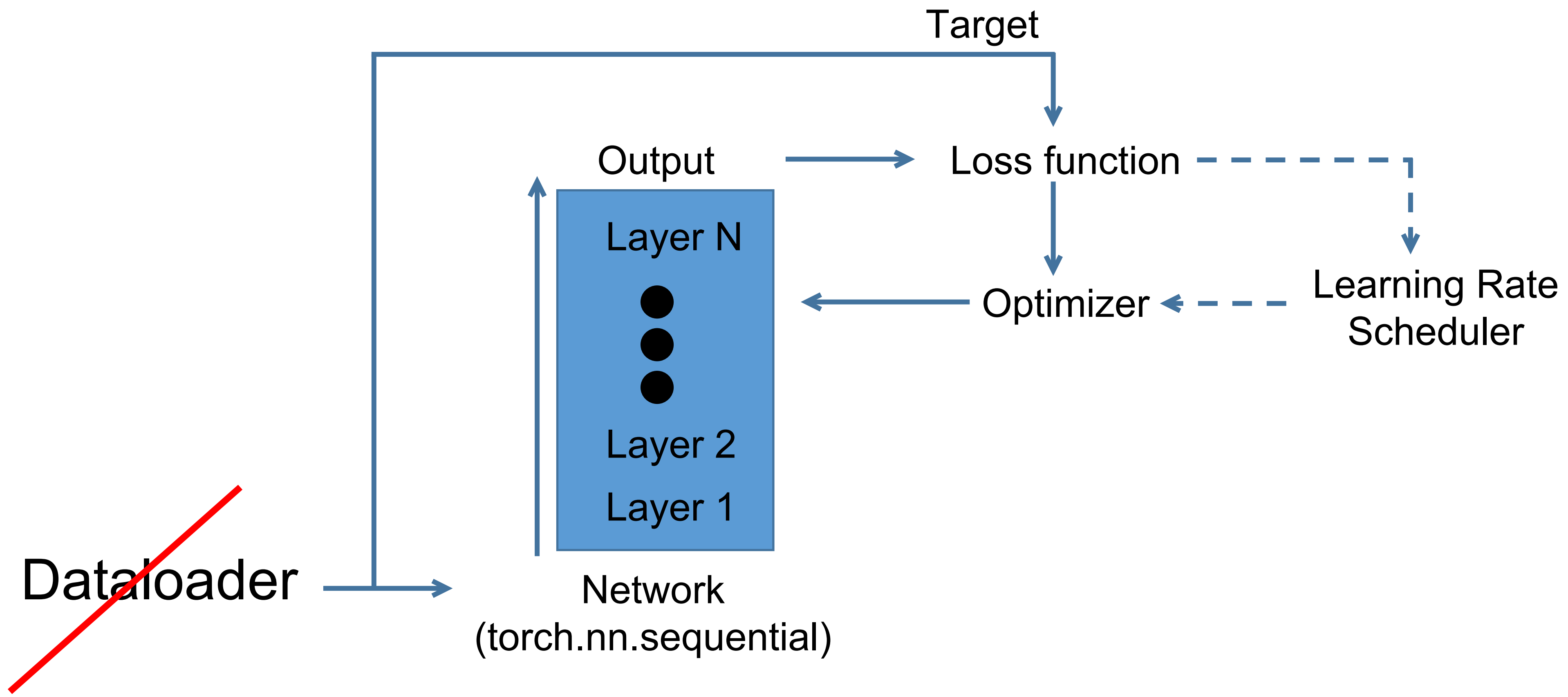
- ASTON ZHANG
- ZACHARY C. LIPTON
- MU LI
- ALEXANDER J. SMOLA

https://d2l.ai/d2l-en.pdf

Fig. 1.3.3 A donkey, a dog, a cat, and a rooster.

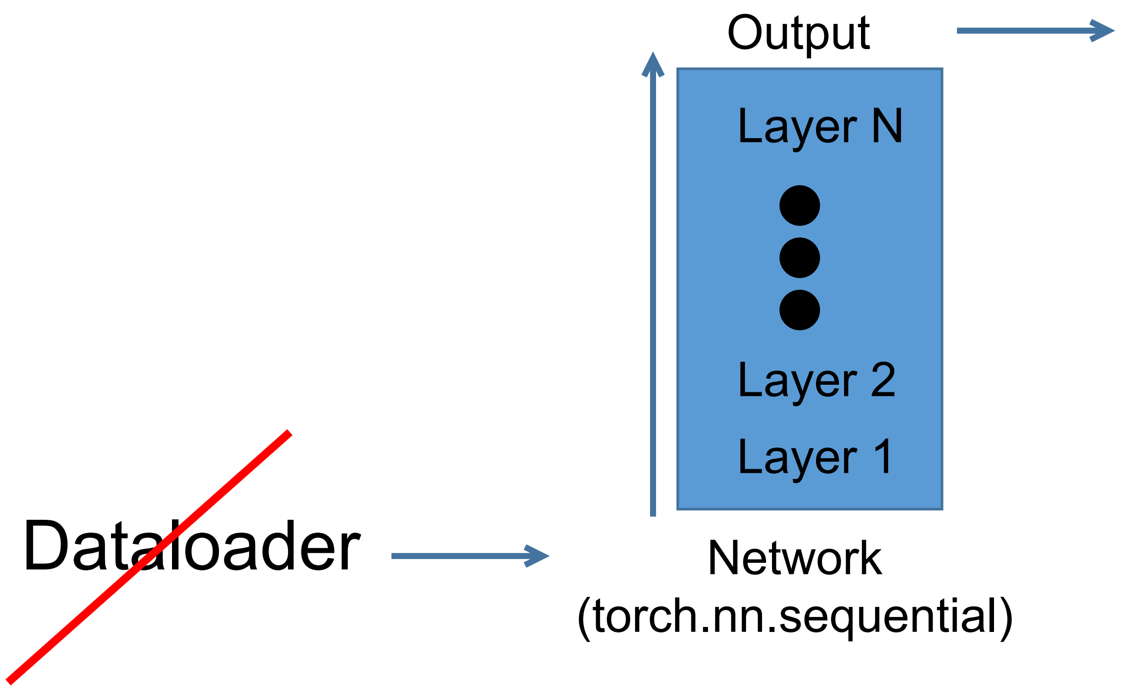
Anatomy of a PyTorch Network + Support

Training of the weights



Anatomy of a PyTorch Network + Support

Inference

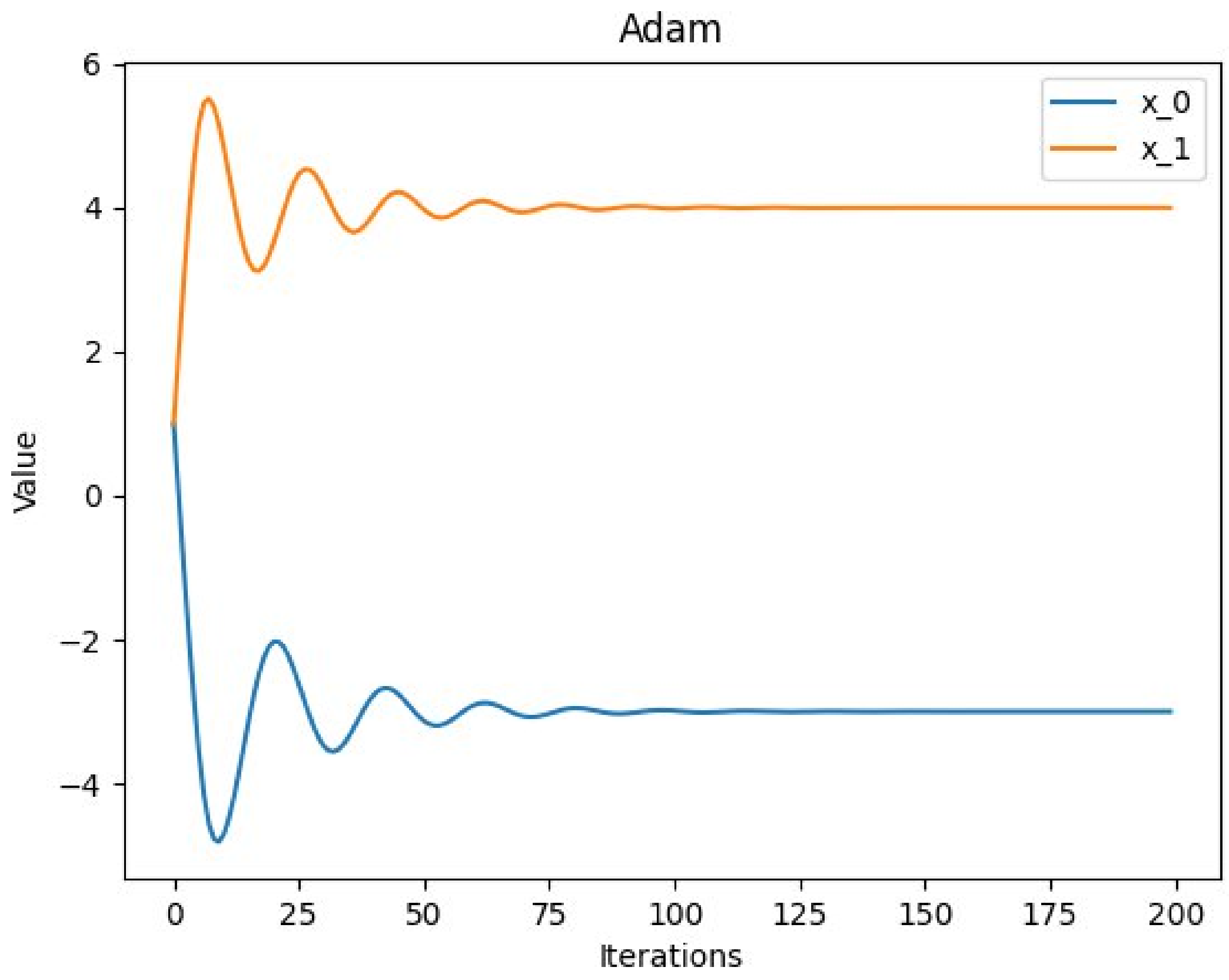


Optimizer & Learning Rate Scheduler

Optimize an equation by hand

$$\begin{aligned} f(x_0, x_1) &= (x_0 + 3)^2 + (x_1 - 4)^2 \\ \frac{\partial f}{\partial x_0} &= 2(x_0 + 3) \\ \frac{\partial f}{\partial x_1} &= 2(x_1 - 4) \end{aligned}$$

Optimize an equation by hand



```
import torch

parameter: torch.nn.Parameter = torch.nn.Parameter(
    torch.ones((2,), dtype=torch.float32)
)
optimizer = torch.optim.Adam([parameter], lr=1)

max_range: int = 200
values: torch.Tensor = torch.zeros(
    (max_range, 2), dtype=parameter.data.dtype, device=parameter.data.device
)
for i in range(0, max_range):
    optimizer.zero_grad()
    parameter.grad = torch.tensor(
        [2 * (parameter.data[0] + 3), 2 * (parameter.data[1] - 4)],
        dtype=parameter.data.dtype,
        device=parameter.data.device,
    )
    values[i, 0] = parameter.data[0]
    values[i, 1] = parameter.data[1]

    optimizer.step()
```

Remember: A parameter has .data and .grad !

Optimizer

Optimizer

Optimizer

Plot the results

```
import matplotlib.pyplot as plt

plt.plot(values[:, 0].detach().cpu().numpy(), label="x_0")

plt.plot(values[:, 1].detach().cpu().numpy(), label="x_1")

plt.title("Adam")
plt.legend()
plt.xlabel("Iterations")
plt.ylabel("Value")
plt.show()
```

Optimizers

There are (too) many different optimizers available.

However, typically Adam and SGD can deal with most of the usual cases.

SGD

```
torch.optim.SGD(params, lr=0.001, momentum=0, dampening=0, weight_decay=0,
nesterov=False, *, maximize=False, foreach=None, differentiable=False,
fused=None)
```

Implements stochastic gradient descent (optionally with momentum).

Adam

```
torch.optim.Adam(params, lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight_decay=0,
amsgrad=False, *, foreach=None, maximize=False, capturable=False,
differentiable=False, fused=None)
```

Implements Adam algorithm.

Adam: A Method for Stochastic Optimization

Save and restore the optimizer state

```
# Save the state
optimizer_state = optimizer.state_dict()
print(optimizer.state_dict()["state"][0]["step"]) # tensor(200.)

# Destroy the state
optimizer = torch.optim.Adam([parameter], lr=1)
print(optimizer.state_dict()["state"]) # {}

# Restore the state
optimizer.load_state_dict(optimizer_state)
print(optimizer.state_dict()["state"][0]["step"]) # tensor(200.)
```

```
step(closure=None)
```

Perform a single optimization step.

```
zero_grad(set_to_none=True)
```

Reset the gradients of all optimized torch.Tensors.

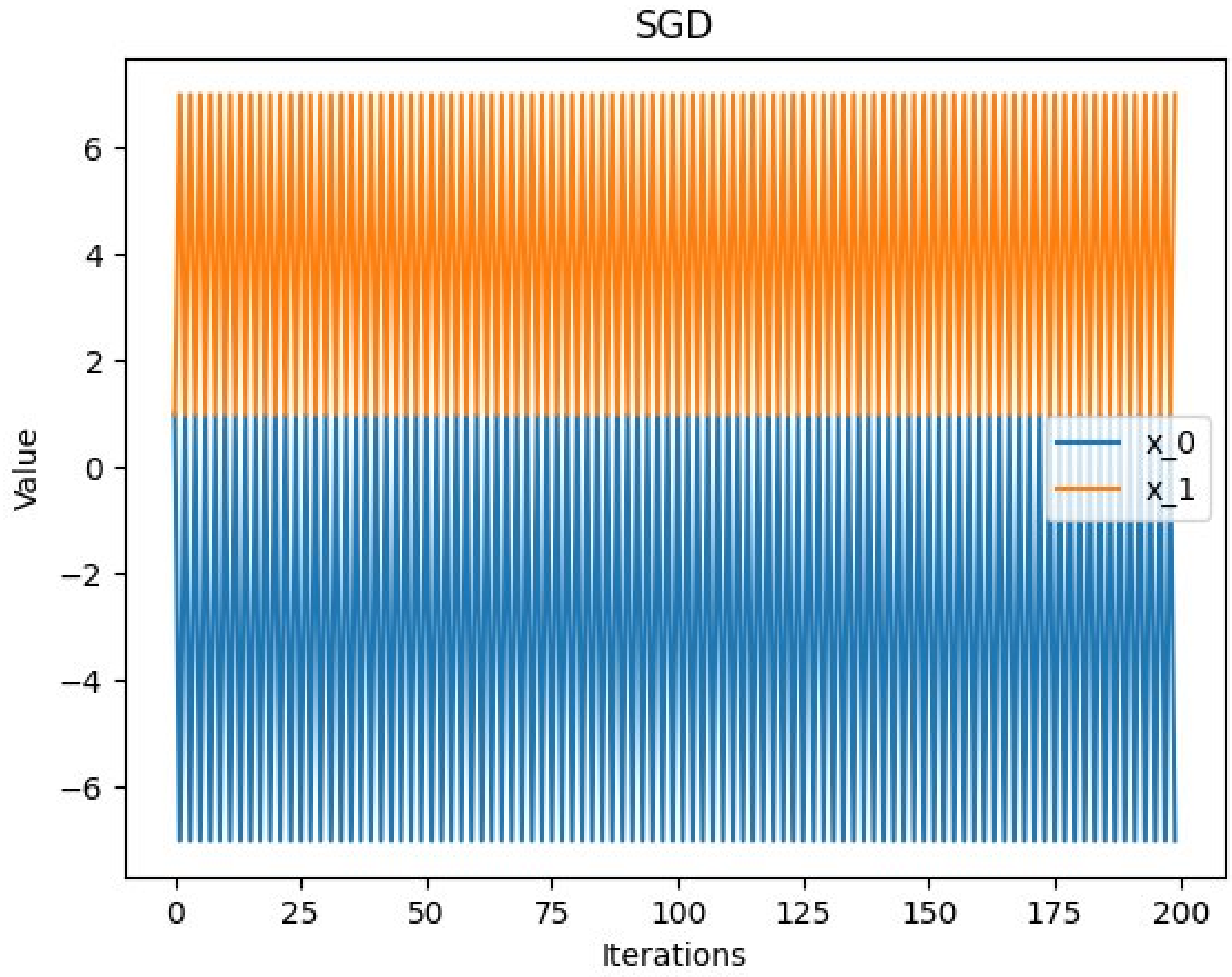
Per-parameter options

“Optimizers also support specifying per-parameter options.”
(i.e. different learning rate values for different sets of parameter)

With one set of parameters, we can readout the learning rate via:

```
optimizer.param_groups[-1]["lr"]
```

Optimizer fails due to a too large learning rate?



Learning Rate Scheduler

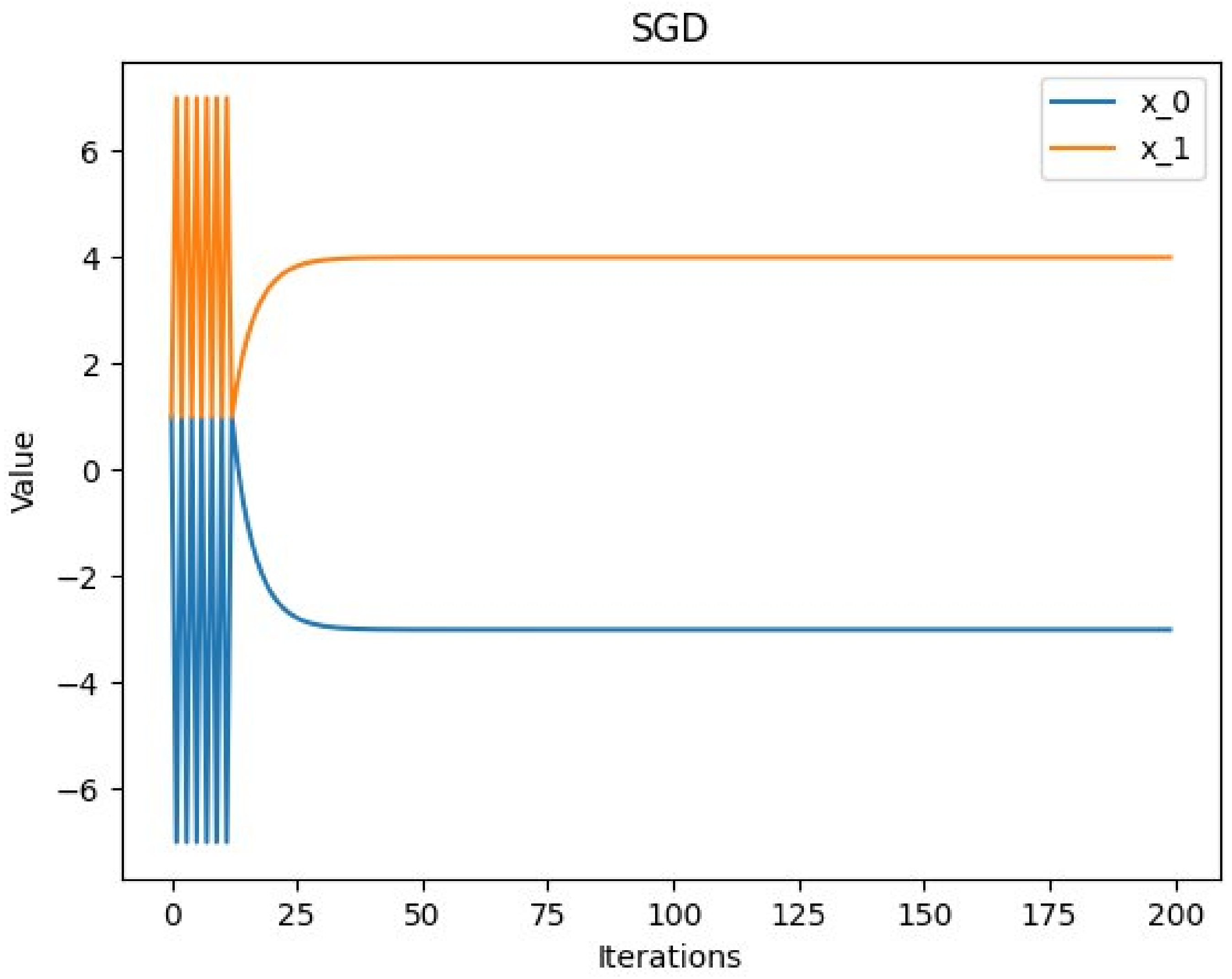
Connecting a learning rate scheduler to the optimizer

```
optimizer = torch.optim.SGD([parameter], lr=1)
lr_scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer)
```

Feeding the quality (here the value of f(x_0,x_1)) of the calculation into the lr scheduler

```
optimizer.step()
lr_scheduler.step((parameter.data[0] + 3) ** 2 + (parameter.data[1] - 4) ** 2)
```

SGD works now



Two types of LR scheduler

With feedback

ReduceLROnPlateau

```
torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.1,
patience=10, threshold=0.0001, threshold_mode='rel', cooldown=0, min_lr=0,
eps=1e-08, verbose='deprecated')
```

```
step(metrics, epoch=None)
```

“Perform a step.”

Without feedback

CosineAnnealingLR

```
torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max, eta_min=0.0,
last_epoch=-1, verbose='deprecated')
```

```
step(epoch=None)
```

“Perform a step.”

Getting the learning rate

```
get_last_lr()
```

“Return last computed learning rate by current scheduler.”

```
get_lr()
```

“Compute learning rate using chainable form of the scheduler.”

I got a “NotImplementedError” for this one.

Or read it out from the optimizer:

```
optimizer.param_groups[-1]["lr"]
```

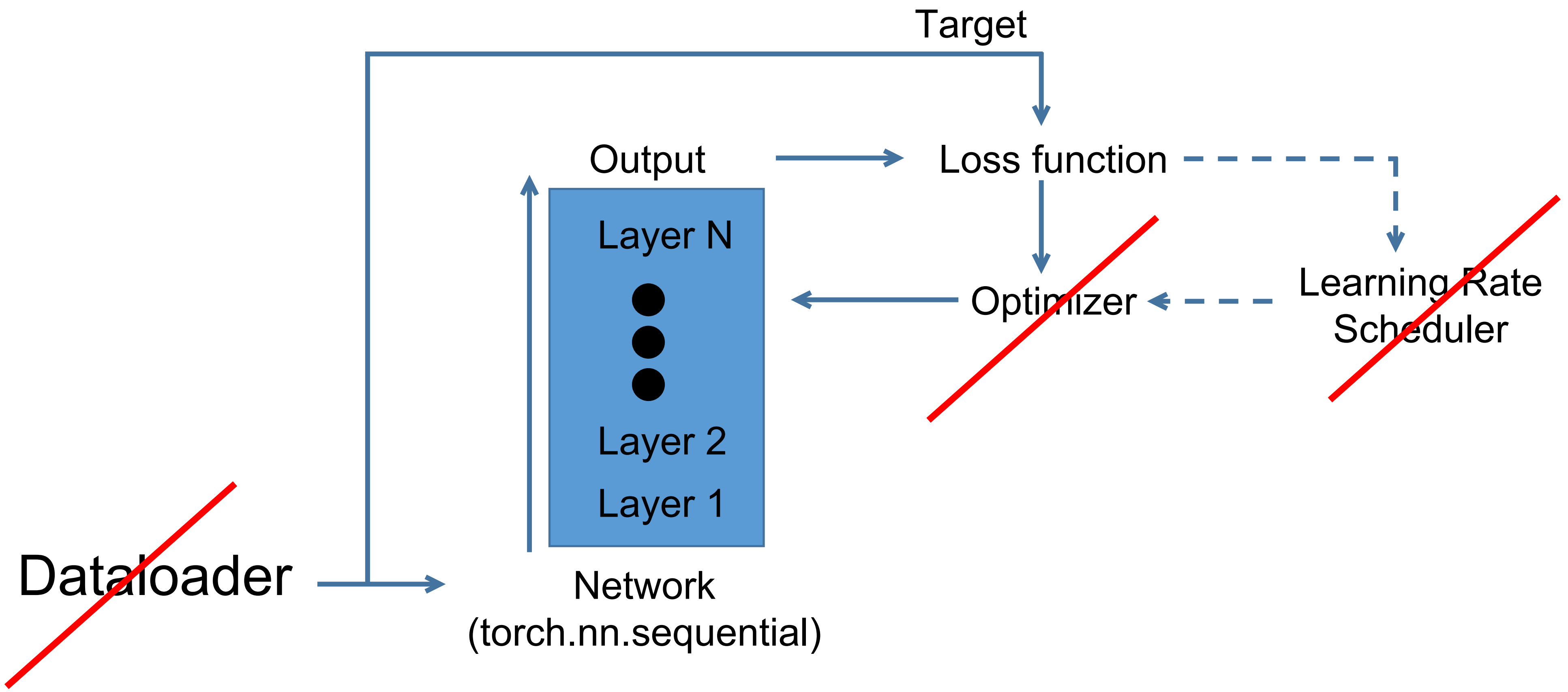
Types of LR Scheduler

There are many different LR Schedulers

Find a list here.

Anatomy of a PyTorch Network + Support

Training of the weights



Tensorboard

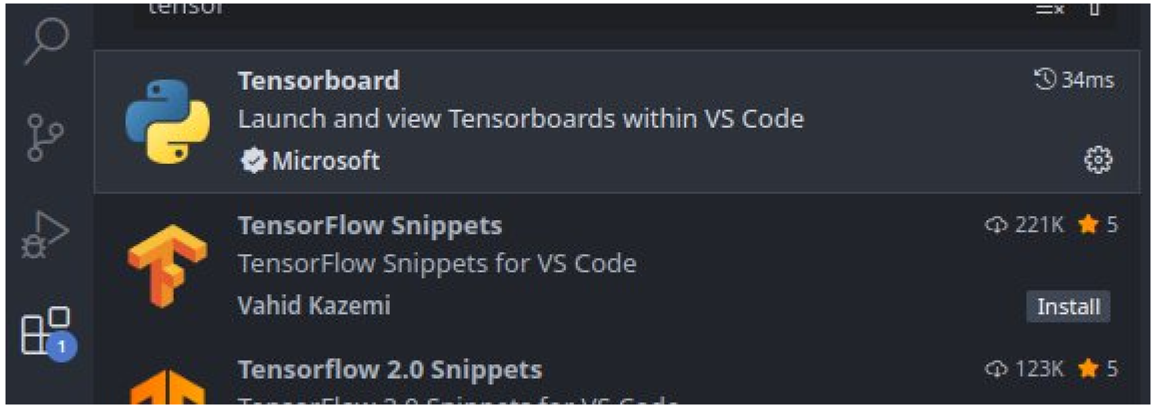
Tensorboard is for logging the progress of training your network.

While Tensorboard is fully open source,
there also exists the freemium service Weights & Biases.

Tensorboard

pip install tensorboard

For VS Code install the Tensorflow extension

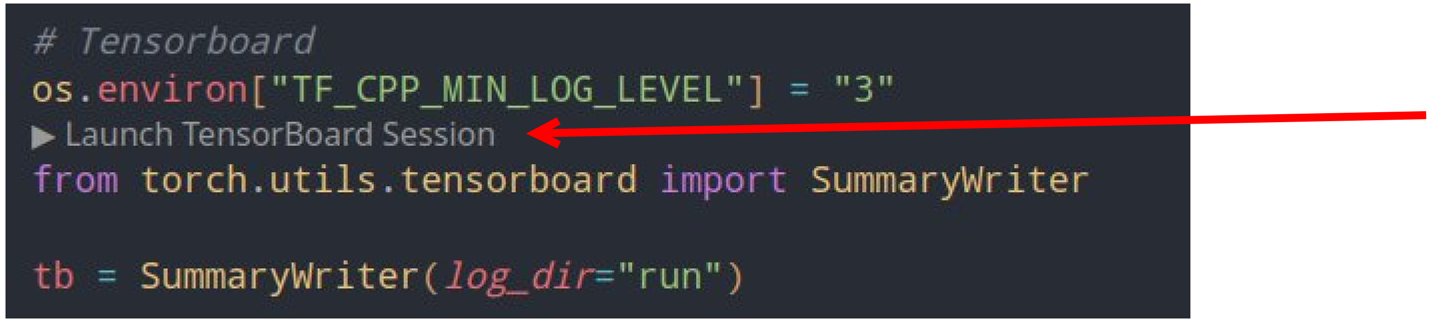

tensor

Tensorboard
Launch and view Tensorboards within VS Code
Microsoft
34ms

TensorFlow Snippets
TensorFlow Snippets for VS Code
Vahid Kazemi
221K 5
Install

Tensorflow 2.0 Snippets
123K 5

Can be used in the editor like this:


```
# Tensorboard
os.environ["TF_CPP_MIN_LOG_LEVEL"] = "3"
▶ Launch TensorBoard Session
from torch.utils.tensorboard import SummaryWriter

tb = SummaryWriter(log_dir="run")
```

torch.utils.tensorboard

```
torch.utils.tensorboard.writer.SummaryWriter(log_dir=None, comment='',
purge_step=None, max_queue=10, flush_secs=120, filename_suffix='')
```

“Writes entries directly to event files in the log_dir to be consumed by TensorBoard.”

```
add_scalar(tag, scalar_value, global_step=None, walltime=None, new_style=False,
double_precision=False)
```

“Add scalar data to summary.”

```
add_histogram(tag, values, global_step=None, bins='tensorflow', walltime=None,
max_bins=None)
```

“Add histogram to summary.”

Also available:

add_scalars, add_image, add_images, add_figure, add_video, add_audio, add_text, add_graph, add_embedding, add_custom_scalars, add_mesh, add_hparams

torch.utils.tensorboard

flush()

“Flushes the event file to disk.

Call this method to make sure that all pending events have been written to disk.”

close()

Closes the SummaryWriter instanz

Simplified example

```
# Tensorboard
os.environ["TF_CPP_MIN_LOG_LEVEL"] = "3"
from torch.utils.tensorboard import SummaryWriter

tb = SummaryWriter(log_dir="run")

[...]

for epoch_id in range(0, number_of_epoch):
    [...]
    tb.add_scalar("Error Test", 100.0 - perfomance_test_correct, epoch_id)
    tb.flush()

[...]

tb.close()
```

How to read out data from Tensorboard files

```
import os
import glob

os.environ["TF_CPP_MIN_LOG_LEVEL"] = "3"
from tensorboard.backend.event_processing

import event_accumulator  # type: ignore
import numpy as np   def get_data(path: str):
    acc = event_accumulator.EventAccumulator(path)
    acc.Reload()
    which_scalar = "Error Test"
    te = acc.Scalars(which_scalar)
    np_temp = np.zeros((len(te), 2))
    for id in range(0, len(te)):
        np_temp[id, 0] = te[id].step
        np_temp[id, 1] = te[id].value
    np_temp = np.nan_to_num(np_temp)
    print(np_temp)
    return np_temp

for path in glob.glob("run*"):
    print(path)
    data = get_data(path)
    np.save("data_" + path + ".npy", data)
```

Loss functions

This is how we measure what we have against what we want.

There are too many loss functions

torch.nn.L1Loss	Creates a criterion that measures the mean absolute error (MAE) between each element in the input
torch.nn.MSELoss	Creates a criterion that measures the mean squared error (squared L2 norm) between each element in the input
torch.nn.CrossEntropyLoss	This criterion computes the cross entropy loss between input logits and target.
torch.nn.CTCLoss	The Connectionist Temporal Classification loss.
torch.nn.NLLLoss	The negative log likelihood loss.
torch.nn.PoissonNLLLoss	Negative log likelihood loss with Poisson distribution of target.
torch.nn.GaussianNLLLoss	Gaussian negative log likelihood loss.
torch.nn.KLDivLoss	The Kullback-Leibler divergence loss.
torch.nn.BCELoss	Creates a criterion that measures the Binary Cross Entropy between the target and the input probabilities:
torch.nn.BCEWithLogitsLoss	This loss combines a Sigmoid layer and the BCELoss in one single class.
torch.nn.MarginRankingLoss	Creates a criterion that measures the loss
torch.nn.HingeEmbeddingLoss	Measures the loss given an input tensor
torch.nn.MultiLabelMarginLoss	Creates a criterion that optimizes a multi-class multi-classification hinge loss (margin-based loss)
torch.nn.HuberLoss	Creates a criterion that uses a squared term if the absolute element-wise error falls below delta and a delta-scaled L1 term otherwise.
torch.nn.SmoothL1Loss	Creates a criterion that uses a squared term if the absolute element-wise error falls below beta and an L1 term otherwise.
torch.nn.SoftMarginLoss	Creates a criterion that optimizes a two-class classification logistic loss
torch.nn.MultiLabelSoftMarginLoss	Creates a criterion that optimizes a multi-label one-versus-all loss based on max-entropy
torch.nn.CosineEmbeddingLoss	Creates a criterion that measures the loss given input tensors and a Tensor label
torch.nn.MultiMarginLoss	Creates a criterion that optimizes a multi-class classification hinge loss (margin-based loss)
torch.nn.TripletMarginLoss	Creates a criterion that measures the triplet loss given an input tensors
torch.nn.TripletMarginWithDistanceLoss	Creates a criterion that measures the triplet loss given input tensors

CrossEntropyLoss

```
torch.nn.CrossEntropyLoss(weight=None, size_average=None, ignore_index=-100,
reduce=None, reduction='mean', label_smoothing=0.0)
```

“This criterion computes the cross entropy loss between input logits and target.”

```
# Example of target with class indices
loss = nn.CrossEntropyLoss()
input = torch.randn(3, 5, requires_grad=True)
target = torch.empty(3, dtype=torch.long).random_(5)
output = loss(input, target)
output.backward()

# Example of target with class probabilities
loss = nn.CrossEntropyLoss()
input = torch.randn(3, 5, requires_grad=True)
target = torch.randn(3, 5).softmax(dim=1)
output = loss(input, target)
output.backward()
```

$$l_n = -\sum_{c=1}^{C} w_c \log \frac{\exp(x_{n,c})}{\sum_{i=1}^{C} \exp(x_{n,i})} y_{n,c}$$

MSELoss

```
torch.nn.CrossEntropyLoss(weight=None, size_average=None, ignore_index=-100,
reduce=None, reduction='mean', label_smoothing=0.0)
```

Creates a criterion that measures the mean squared error (squared L2 norm) between each element in the input x and target y.

```
loss = nn.MSELoss()
input = torch.randn(3, 5, requires_grad=True)
target = torch.randn(3, 5)
output = loss(input, target)
output.backward()
```

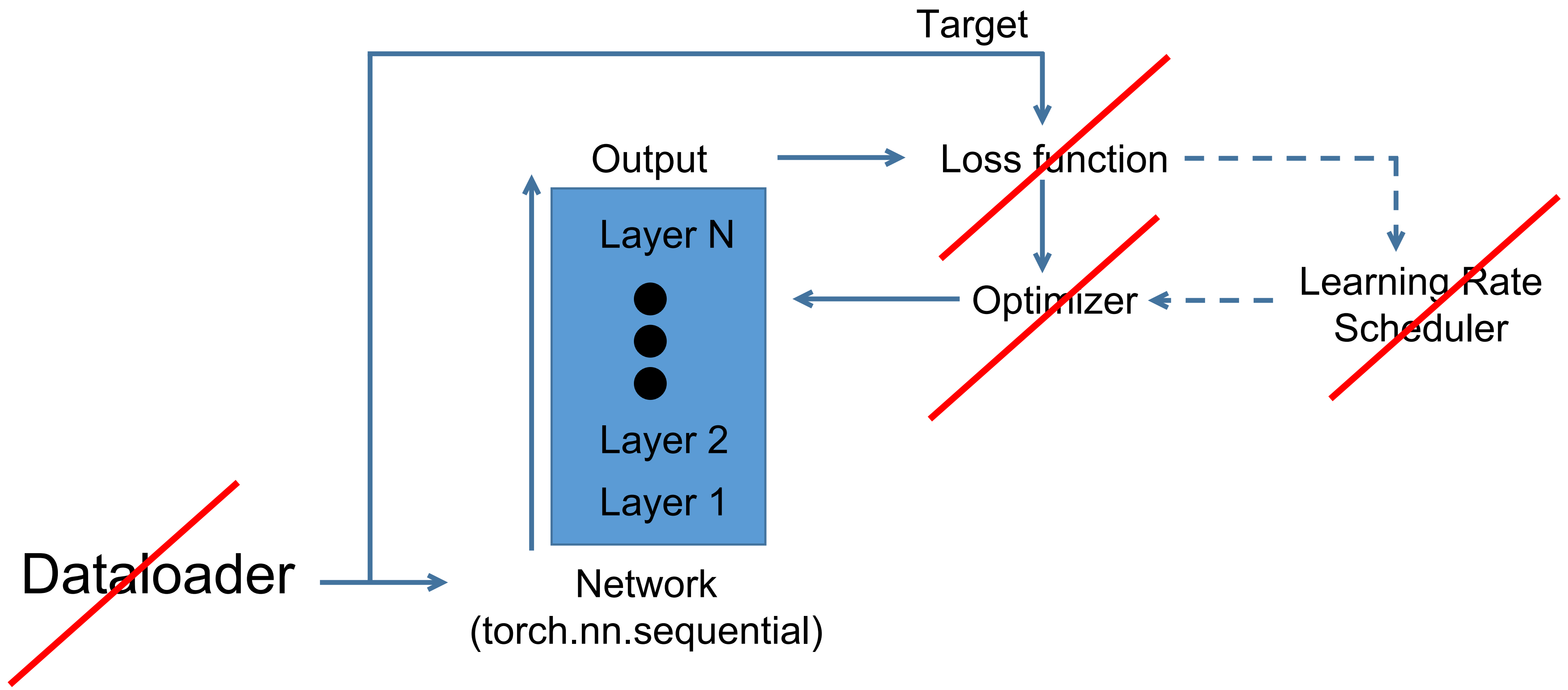
$$l_n = (x_n - y_n)^2$$

Convert labels in one hot encoding (scatter)

```
number_of_output_neurons = 10
# Convert label into one hot
target_one_hot: torch.Tensor = torch.zeros(
    (
        target.shape[0],
        number_of_output_neurons,
    ),
    device=target.device,
    dtype=target.dtype,
)
target_one_hot.scatter_(
    1,
    target.unsqueeze(1),
    torch.ones(
        (target.shape[0], 1),
        device=target.device,
        dtype=target.dtype,
    ),
)
```

Anatomy of a PyTorch Network + Support

Training of the weights



What makes a layer tick...

Internals of Linear Layer

```python
import torch
import math
```

Is a child of Module

```python
class Linear(torch.nn.Module):
    __constants__ = ["in_features", "out_features"]
    in_features: int
    out_features: int
    weight: torch.Tensor
```

Information for just-in-time compilation?

This is probably wrong and should be torch.nn.Parameter

```
def __init__(
    self,
    in_features: int,
    out_features: int,
    bias: bool = True,
    device=None,
    dtype=None,
) -> None:
    factory_kwargs = {"device": device, "dtype": dtype}
    super().__init__()
    self.in_features = in_features
    self.out_features = out_features
    self.weight = torch.nn.Parameter(
        torch.empty((out_features, in_features), **factory_kwargs)
    )
    if bias:
        self.bias = torch.nn.Parameter(torch.empty(out_features, **factory_kwargs))
    else:
        self.register_parameter("bias", None)
    self.reset_parameters()
```

Weight

Bias

Parameter initialization

Parameter initialization

```
def reset_parameters(self) -> None:
    torch.nn.init.kaiming_uniform_(self.weight, a=math.sqrt(5))
    if self.bias is not None:
        fan_in, _ = torch.nn.init._calculate_fan_in_and_fan_out(self.weight)
        bound = 1 / math.sqrt(fan_in) if fan_in > 0 else 0
        torch.nn.init.uniform_(self.bias, -bound, bound)
```

Data processing!!!

```
def forward(self, input: torch.Tensor) -> torch.Tensor:
    return torch.nn.functional.linear(input, self.weight, self.bias)
```

Pretty printing

```
def extra_repr(self) -> str:
    return (
        f"in_features={self.in_features}, "
        "out_features={self.out_features}, "
        "bias={self.bias is not None}"        )
```

The information processing in the layer

```
    def forward(self, input: torch.Tensor) -> torch.Tensor:
        return torch.nn.functional.linear(input, self.weight, self.bias)
```

torch.nn.functional.linear

```
torch.nn.functional.linear(input, weight, bias=None) → Tensor
```

“Applies a linear transformation to the incoming data:”

$$y = xA^T + b$$

We never see what backwards is doing and it is defined by torch.nn.functional.linear.

Extending torch.autograd

Example — Linear Layer

```
class FunctionalLinear(torch.autograd.Function):
    @staticmethod
    def forward(  # type: ignore
        ctx, input: torch.Tensor, weight: torch.Tensor, bias: torch.Tensor
    ) -> torch.Tensor:
        output = (weight.unsqueeze(0) * input.unsqueeze(1)).sum(dim=-1)

        if bias is not None:
            output = output + bias.unsqueeze(0)

        # #####################################
        # Save the necessary data for the backward pass        #
        #####################################
        ctx.save_for_backward(
            input,
            weight,
            bias,
        )
        return output
```

Normal forward calculation

Storing the data for backward

```
@staticmethod
@torch.autograd.function.once_differentiable
 def backward(  # type: ignore
    ctx, grad_output: torch.Tensor
 ) -> tuple[torch.Tensor | None, torch.Tensor | None, torch.Tensor | None]:

    # ########################################
    # Get the variables back
    # ########################################
    (
        input,
        weight,
        bias,
    ) = ctx.saved_tensors
```

All arguments of forward (except ctx) require a return value.

Retrieving the stored data for backward

```
# ##########################################
# Default output
# ##########################################
grad_input: torch.Tensor | None = None
grad_weight: torch.Tensor | None = None
grad_bias: torch.Tensor | None = None

grad_weight = grad_output.unsqueeze(-1) * input.unsqueeze(-2)

if bias is not None:
    grad_bias = grad_output.detach().clone()

grad_input = (grad_output.unsqueeze(-1) * weight.unsqueeze(0)).sum(dim=1)

return grad_input, grad_weight, grad_bias
```

All arguments of forward (except ctx) require a return value. Default is None

Hopefully correct gradients for weight & bias

Hopefully correct new back-propagating error

All arguments of forward (except ctx) require a return value.

How to include it into your layer

```
class MyOwnLayer(torch.nn.Module):
    def __init__([...]):

        [...]
        self.reset_parameters()

        self.functional_linear = FunctionalLinear.apply

    def forward(
        self,
        input: torch.Tensor,
    ) -> torch.Tensor:
        return self.functional_linear(input, self.weight, self.bias)
```

! **Not everything needs to be in our own autograd function. In fact, try to put as little as possible into your own autograd function and let the rest handle by torch's autograd. Less is more.**

Extending torch.autograd — [When to use]

Examples are approximations for backwards when spikes are used for forward or C++ modules.

“In general, implement a custom function if you want to perform computations in your model that are not differentiable or rely on non-PyTorch libraries (e.g., NumPy), but still wish for your operation to chain with other ops and work with the autograd engine.

In some situations, custom functions can also be used to improve performance and memory usage: If you implemented your forward and backward passes using a C++ extension, you can wrap them in Function to interface with the autograd engine. If you’d like to reduce the number of buffers saved for the backward pass, custom functions can be used to combine ops together.”

Extending torch.autograd — How to use

You'll need to define two methods:

"**forward()** is the code that performs the operation. **It can take as many arguments as you want, with some of them being optional, if you specify the default values.** All kinds of Python objects are accepted here. Tensor arguments that track history (i.e., with requires_grad=True) will be converted to ones that don't track history before the call, and their use will be registered in the graph. Note that this logic won't traverse lists/dicts/any other data structures and will only consider tensors that are direct arguments to the call. You can return either a single Tensor output, or a tuple of tensors if there are multiple outputs.

backward() (or vjp()) defines the gradient formula. It will be given as many Tensor arguments as there were outputs, with each of them representing gradient w.r.t. that output. It is important NEVER to modify these in-place. **It should return as many tensors as there were inputs, with each of them containing the gradient w.r.t. its corresponding input. If your inputs didn't require gradient** (needs_input_grad is a tuple of booleans indicating whether each input needs gradient computation), **or were non-Tensor objects, you can return python:None.** Also, if you have optional arguments to forward() you can return more gradients than there were inputs, as long as they're all None."

Extending torch.autograd — How to use

“It is your responsibility to use the functions in ctx properly in order to ensure that the new Function works properly with the autograd engine. save_for_backward() must be used to save any tensors to be used in the backward pass. Non-tensors should be stored directly on ctx. If tensors that are neither input nor output are saved for backward your Function may not support double backward”

Extending torch.autograd — How to use

I never had to use these:

“mark_dirty() must be used to mark any input that is modified inplace by the forward function.

mark_non_differentiable() must be used to tell the engine if an output is not differentiable. By default all output tensors that are of differentiable type will be set to require gradient. Tensors of non-differentiable type (i.e., integral types) are never marked as requiring gradients.

set_materialize_grads() can be used to tell the autograd engine to optimize gradient computations in the cases where the output does not depend on the input by not materializing grad tensors given to backward function. That is, if set to False, None object in Python or “undefined tensor” (tensor x for which x.defined() is False) in C++ will not be converted to a tensor filled with zeros prior to calling backward, and so your code will need to handle such objects as if they were tensors filled with zeros. The default value of this setting is True.”

Extending torch.autograd — How to use

“If your Function does not support double backward you should explicitly declare this by decorating backward with the once_differentiable(). With this decorator, attempts to perform double backward through your function will produce an error. See our double backward tutorial for more information on double backward.”

TORCH.AUTOGRAD.FUNCTION.FUNCTIONCTX.SAVE_FOR_BACKWARD

```
FunctionCtx.save_for_backward(*tensors)
```

“Saves given tensors for a future call to backward().

save_for_backward should be called at most once, only from inside the forward() method, and only with tensors.

All tensors intended to be used in the backward pass should be saved with save_for_backward (as opposed to directly on ctx) to prevent incorrect gradients and memory leaks, and enable the application of saved tensor hooks.

Note that if intermediary tensors, tensors that are neither inputs nor outputs of forward(), are saved for backward, your custom Function may not support double backward. Custom Functions that do not support double backward should decorate their backward() method with @once_differentiable so that performing double backward raises an error. If you’d like to support double backward, you can either recompute intermediaries based on the inputs during backward or return the intermediaries as the outputs of the custom Function. See the double backward tutorial for more details.

In backward(), saved tensors can be accessed through the saved_tensors attribute. Before returning them to the user, a check is made to ensure they weren’t used in any in-place operation that modified their content.

Arguments can also be None. This is a no-op.”

Save

```
ctx.save_for_backward(x, y, w, out)
```

Non-tensor (e.g. int):

```
ctx.z = z
```

Access

```
x, y, w, out = ctx.saved_tensors
```

Non-tensor (e.g. int):

```
z = ctx.z
```

Converting a function into a layer

Lambda

```
torchvision.transforms.Lambda(lambd)
```

“Apply a user-defined lambda as a transform. This transform does not support torchscript.”

```
import torch
import torchvision.transforms as transforms  # type: ignore

# Permute Example
permute_layer = transforms.Lambda(lambda x: torch.permute(x, dims=(0, 2, 3, 1)))
input = torch.zeros((10, 11, 12, 13))
output = permute_layer(input)
print(input.shape) # torch.Size([10, 11, 12, 13])
print(output.shape) # torch.Size([10, 12, 13, 11])
print(permute_layer) # Lambda()
```

Combining layers into a network

Theoretically there is **no need** to combine layers into a network but it is much nice to use it as a network

Option 1: a custom class based on torch.nn.Module

```
import torch
class Network(torch.nn.Module):
    def __init__(
        self,
        input_number_of_channel: int,
        number_of_output_channels_conv1: int,
        kernel_size_conv1: int,
        stride_conv1: int,
        padding_conv1: int,
        kernel_size_pool1: int,
        stride_pool1: int,
        padding_pool1: int,
    ) -> None:
        super().__init__()
        self.layer_1 = torch.nn.Conv2d(
            in_channels=input_number_of_channel,
            out_channels=number_of_output_channels_conv1,
            kernel_size=kernel_size_conv1,
            stride=stride_conv1,
            padding=padding_conv1,
        )
        self.layer_2 = torch.nn.ReLU()
        self.layer_3 = torch.nn.MaxPool2d(
            kernel_size=kernel_size_pool1, stride=stride_pool1, padding=padding_pool1
        )
```

__init__()

Define the layers as attributes of the class

Option 1: a custom class based on torch.nn.Module

```
def forward(self, inputs: torch.Tensor) -> torch.Tensor:
  output = self.layer_1(inputs)
  output = self.layer_2(output)
  output = self.layer_3(output)
  return output
```

forward()

Feeding the data through the layers.

Option 2: torch.nn.Sequential

```
torch.nn.Sequential(*args: Module)
```

“A sequential container.”

```
network = torch.nn.Sequential(
    torch.nn.Conv2d(
        in_channels=input_number_of_channel,
        out_channels=number_of_output_channels_conv1,
        kernel_size=kernel_size_conv1,
        stride=stride_conv1,
        padding=padding_conv1,
    ),
    torch.nn.ReLU(),
    torch.nn.MaxPool2d(
        kernel_size=kernel_size_pool1,
        stride=stride_pool1,
        padding=padding_pool1,
    ),
)
```

Define the layers

Option 1 vs Option 2?

I prefer option 2 (torch.nn.Sequential) and use it where possible.

In the moment, option 1 is still necessary for skip connection or branching passes.

I developed and suggested a simple working extension for PyTorch called SequentialSplit but I failed due to politics.

Nevertheless, you can mix both options.

More details for torch.nn.Sequential

```
network = torch.nn.Sequential(
    torch.nn.Conv2d(
        in_channels=input_number_of_channel,
        out_channels=number_of_output_channels_conv1,
        kernel_size=kernel_size_conv1,
        stride=stride_conv1,
        padding=padding_conv1,
    ),
    torch.nn.ReLU(),
    torch.nn.MaxPool2d(
        kernel_size=kernel_size_pool1,
        stride=stride_pool1,
        padding=padding_pool1,
    ),
)

print(network)
```

```
Sequential(
  (0): Conv2d(1, 32, kernel_size=(5, 5), stride=(1, 1))
  (1): ReLU()
  (2): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), padding=0, dilation=1, ceil_mode=False)
)
```

Dynamically creating the network

```
network = torch.nn.Sequential()

network.append(
    torch.nn.Conv2d(
        in_channels=input_number_of_channel,
        out_channels=number_of_output_channels_conv1,
        kernel_size=kernel_size_conv1,
        stride=stride_conv1,
        padding=padding_conv1,
    )
)
network.append(torch.nn.ReLU())

network.append(
    torch.nn.MaxPool2d(
        kernel_size=kernel_size_pool1,
        stride=stride_pool1,
        padding=padding_pool1,
    )
)
```

Append...

Append...

Append...

Labelling layers with add_module

```
network = torch.nn.Sequential()
network.add_module(
    "Conv2d Layer One",
    torch.nn.Conv2d(
       in_channels=input_number_of_channel,
       out_channels=number_of_output_channels_conv1,
       kernel_size=kernel_size_conv1,
       stride=stride_conv1,
       padding=padding_conv1,
    ),
)
network.append(torch.nn.ReLU())
network.append(
    torch.nn.MaxPool2d(
       kernel_size=kernel_size_pool1,
       stride=stride_pool1,
       padding=padding_pool1,
    )
)
```

Use add_module() instead of append()

```
Sequential(
  (Conv2d Layer One): Conv2d(1, 32, kernel_size=(5, 5), stride=(1, 1))
  (1): ReLU()
  (2): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), padding=0, dilation=1, ceil_mode=False)
)
```

Lazy layers vs. Intelligent approach

Linear ? (same for the flatten layer!)

```
torch.nn.Linear(in_features, out_features, bias=True, device=None, dtype=None)
```

“Applies an affine linear transformation to the incoming data”

LazyLinear

```
torch.nn.LazyLinear(out_features, bias=True, device=None, dtype=None)
```

“A torch.nn.Linear module where in_features is inferred.”

However for many research operations, like manipulating weights before the first use of the network, the lazy layer class has limits.

Intelligent approach

```
fake_input: torch.Tensor = torch.zeros((1, 3, 28, 28))

network = torch.nn.Sequential() network.append(
    torch.nn.Conv2d(
        in_channels=input_number_of_channel,
        out_channels=number_of_output_channels_conv1,
        kernel_size=kernel_size_conv1,
        stride=stride_conv1,
        padding=padding_conv1,
    )
)
print("Pre Conv2d: ", fake_input.shape)
fake_input = network[-1](fake_input)
print("Post Conv2d: ", fake_input.shape)
print()
```

We create a fake input based on the dimensions for the real input with one pattern in the batch dimension

We feed that data through the last appended layer i.e. network[-1]

```
Pre Conv2d:  torch.Size([1, 3, 28, 28])
Post Conv2d:  torch.Size([1, 32, 24, 24])
```

Intelligent approach

```
print("Pre Conv2d: ", fake_input.shape)
fake_input = network[-1](fake_input)
print("Post Conv2d: ", fake_input.shape)
print()
```

```
Pre ReLu:  torch.Size([1, 32, 24, 24])
Post ReLu:  torch.Size([1, 32, 24, 24])
```

Intelligent approach

```
network.append(
    torch.nn.MaxPool2d(
        kernel_size=kernel_size_pool1,
        stride=stride_pool1,
        padding=padding_pool1,
    )
)
print("Pre MaxPool2d: ", fake_input.shape)
fake_input = network[-1](fake_input)
print("Post MaxPool2d: ", fake_input.shape)
print()
```

```
Pre MaxPool2d:  torch.Size([1, 32, 24, 24])
Post MaxPool2d:  torch.Size([1, 32, 12, 12])
```

Inspecting the network

```
print(network.__dict__)
```

```
{'training': True, '_parameters': {}, '_buffers': {}, '_non_persistent_buffers_set': set(),
'_backward_pre_hooks': OrderedDict(), '_backward_hooks': OrderedDict(),
'_is_full_backward_hook': None, '_forward_hooks': OrderedDict(), '_forward_hooks_with_kwargs':
OrderedDict(), '_forward_hooks_always_called': OrderedDict(), '_forward_pre_hooks':
OrderedDict(), '_forward_pre_hooks_with_kwargs': OrderedDict(), '_state_dict_hooks':
OrderedDict(), '_state_dict_pre_hooks': OrderedDict(), '_load_state_dict_pre_hooks':
OrderedDict(), '_load_state_dict_post_hooks': OrderedDict(), '_modules': {
            '0': Conv2d(3, 32, kernel_size=(5, 5), stride=(1, 1)),
            '1': ReLU(),
            '2': MaxPool2d(kernel_size=(2, 2), stride=(2, 2), padding=0, dilation=1, ceil_mode=False)
            }
}
```

Or use this for the architecture part of the network:

```
print(network.__dict__["_modules"])
```

Save / Load the **whole** network

“whole” means with the structure of the network.

```
torch.save(network, "torch_network.pt")
```

```
network = torch.load("torch_network.pt", weights_only=False)
network.eval()
```

Save / Load the **weights** of the network

```
torch.save(network.state_dict(), "torch_network_dict.pt")
```

1. Re-create the network
2. Replace the weights with the stored weights

```
network.load_state_dict(torch.load("torch_network_dict.pt", weights_only=True))
network.eval()
```

A closer look into our layers

```
for layer_id, layer in enumerate(network):
    print(f"Layer ID: {layer_id}")
    print(layer)
    # or:
    # print(network[layer_id])
    print()
```

```
Layer ID: 0
Conv2d(3, 32, kernel_size=(5, 5), stride=(1, 1))

Layer ID: 1
ReLU()

Layer ID: 2
MaxPool2d(kernel_size=(2, 2), stride=(2, 2), padding=0, dilation=1, ceil_mode=False)
```

Extracting the activities of the network

```
activity: list[torch.Tensor] = []
activity.append(input)            The initial input goes here...

for layer in network:
        activity.append(layer(activity[-1]))    Let the network does it thing
                                                Step by step...

for id, data in enumerate(activity):
        print(f"ID: {id} Shape:{data.shape}")   Do something with the collected activities
```

```
ID: 0 Shape:torch.Size([1, 3, 28, 28])     initial input
ID: 1 Shape:torch.Size([1, 32, 24, 24])    Conv2d
ID: 2 Shape:torch.Size([1, 32, 24, 24])    ReLu
ID: 3 Shape:torch.Size([1, 32, 12, 12])    MaxPool2d
```

Accessing the parameters / weights of a layer

Let us look at layer 0 (Conv2d)

```
print(network[0].__dict__)
```

```
{'training': True, '_parameters': {'weight': Parameter containing:
tensor([[[[-0.0449,  0.1092, -0.1087,  0.0159,  0.0759],
          ...,
          [ 0.0762,  0.0426,  0.0279,  0.0939,  0.0803]]]], requires_grad=True),
'bias': Parameter containing: tensor([ 0.0202, -0.0314, -0.0025,  0.1135,  0.0834,  0.0860,  0.0159,  0.0703,
         ...,
          0.0009,  0.0744, -0.0898,  0.0013, -0.0176, -0.0628,  0.0760, -0.0749],
        requires_grad=True)},
 '_buffers': {}, '_non_persistent_buffers_set': set(), '_backward_pre_hooks': OrderedDict(), '_backward_hooks':
OrderedDict(), '_is_full_backward_hook': None, '_forward_hooks': OrderedDict(),
'_forward_hooks_with_kwargs': OrderedDict(), '_forward_hooks_always_called': OrderedDict(),
'_forward_pre_hooks': OrderedDict(), '_forward_pre_hooks_with_kwargs': OrderedDict(), '_state_dict_hooks':
OrderedDict(), '_state_dict_pre_hooks': OrderedDict(), '_load_state_dict_pre_hooks': OrderedDict(),
'_load_state_dict_post_hooks': OrderedDict(), '_modules': {}, 'in_channels': 3, 'out_channels': 32, 'kernel_size':
(5, 5), 'stride': (1, 1), 'padding': (0, 0), 'dilation': (1, 1), 'transposed': False, 'output_padding': (0, 0), 'groups': 1,
'padding_mode': 'zeros', '_reversed_padding_repeated_twice': (0, 0, 0, 0)}
```

What a chaos...

Accessing the parameters / weights of a layer

```
print(network[0].__dict__.keys())
```

These are available entries:

```
dict_keys(['training', '_parameters', '_buffers', '_non_persistent_buffers_set',
'_backward_pre_hooks', '_backward_hooks', '_is_full_backward_hook',
'_forward_hooks', '_forward_hooks_with_kwargs', '_forward_hooks_always_called',
'_forward_pre_hooks', '_forward_pre_hooks_with_kwargs', '_state_dict_hooks',
'_state_dict_pre_hooks', '_load_state_dict_pre_hooks', '_load_state_dict_post_hooks',
'_modules', 'in_channels', 'out_channels', 'kernel_size', 'stride', 'padding', 'dilation',
'transposed', 'output_padding', 'groups', 'padding_mode',
'_reversed_padding_repeated_twice'])
```

Our main interest is located in _parameters.

Accessing the parameters / weights of a layer

```
print(network[0].__dict__["_parameters"].keys())
```

These are the available entries:

```
dict_keys(['weight', 'bias'])
```

Iterating through the layers

```python
for name, module in network.named_modules():
    print(f"Layer name: {name}, Type: {type(module).__name__}")
```

```
Layer name: , Type: Sequential
Layer name: 0, Type: Conv2d
Layer name: 1, Type: ReLU
Layer name: 2, Type: MaxPool2d
```

```python
for module in network:
    print(f"Type: {type(module).__name__}")
```

```
Type: Conv2d
Type: ReLU
Type: MaxPool2d
```

Iterating through the parameters

```
for name, param in network.named_parameters():
    print(f"Parameter name: {name}, Shape: {param.shape}")
```

```
Parameter name: 0.weight, Shape: torch.Size([32, 3, 5, 5])
Parameter name: 0.bias, Shape: torch.Size([32])
```

```
for param in network.parameters():
    print(f"Shape: {param.shape}")
```

```
Shape: torch.Size([32, 3, 5, 5])
Shape: torch.Size([32])
```

Getting / Setting weights

```
conv1_bias = network[0].__dict__["_parameters"]["bias"].data
conv1_weights = network[0].__dict__["_parameters"]["weight"].data
print(conv1_bias.shape)  # -> torch.Size([32])
print(conv1_weights.shape)  # -> torch.Size([32, 1, 5, 5])
```

```
conv1_bias = network[0].__dict__["_parameters"]["bias"]
conv1_weights = network[0].__dict__["_parameters"]["weight"]
print(conv1_bias.shape)  # -> torch.Size([32])
print(conv1_weights.shape)  # -> torch.Size([32, 1, 5, 5])
```

Is .data useless then?
Well, you don’t need it for getting the data but for setting it!

It is the different between setting the value of the weight and destroying the parameter and breaking the optimizer’s connection to the parameter.

Getting / Setting weights

```
print(conv1_bias.shape)  # -> torch.Size([32])
print(conv1_weights.shape)  # -> torch.Size([32, 1, 5, 5])
```

The order of the dimensions is a bit strange.
It is [Output Channel, Input Channel, Kernel X, Kernel Y] for the 2D convolution layer and [Output Channel, Input Channel] for the linear layer.

```
network[0].__dict__["_parameters"]["bias"].data = 5 * torch.ones(
    (32),
    dtype=network[0].__dict__["_parameters"]["bias"].data.dtype,
    device=network[0].__dict__["_parameters"]["bias"].data.device,
)

network[0].__dict__["_parameters"]["weight"].data = torch.ones(
    (32, 1, 5, 5),
    dtype=network[0].__dict__["_parameters"]["weight"].data.dtype,
    device=network[0].__dict__["_parameters"]["weight"].data.device,
)
```

Feeding the optimizer

Normal way:

```
optimizer = torch.optim.Adam(network.parameters())
```

If you need to do something to the parameters or split them between different optimizers, you can give list of parameters to the optimizer too:

```
parameter_list = []
for para in network.parameters():
    parameter_list.append(para)

optimizer = torch.optim.Adam(parameter_list)
```

Backup to option 1 (own network class)

Getting / Setting weights

This does not work any more:

```
conv1_bias = network[0].__dict__["_parameters"]["bias"].data
conv1_weights = network[0].__dict__["_parameters"]["weight"].data
```

X

Let us hope you remember how you named the attributes which contain the layers:

```
self.layer_1 = torch.nn.Conv2d(
    in_channels=input_number_of_channel,
    out_channels=number_of_output_channels_conv1,
    kernel_size=kernel_size_conv1,
    stride=stride_conv1,
    padding=padding_conv1,
)
self.layer_2 = torch.nn.ReLU()
self.layer_3 = torch.nn.MaxPool2d(
    kernel_size=kernel_size_pool1, stride=stride_pool1, padding=padding_pool1
)
```

Getting / Setting weights

```
conv1_bias = network.layer_1.__dict__["_parameters"]["bias"].data
conv1_weights = network.layer_1.__dict__["_parameters"]["weight"].data
print(conv1_bias.shape)  # -> torch.Size([32])
print(conv1_weights.shape)  # -> torch.Size([32, 1, 5, 5])
```

Getting / Setting weights

```
network.layer_1.__dict__["_parameters"]["bias"].data = 5 * torch.ones(
    (32),
    dtype=network.layer_1.__dict__["_parameters"]["bias"].data.dtype,
    device=network.layer_1.__dict__["_parameters"]["bias"].data.device,
)
network.layer_1.__dict__["_parameters"]["weight"].data = torch.ones(
    (32, 1, 5, 5),
    dtype=network.layer_1.__dict__["_parameters"]["weight"].data.dtype,
    device=network.layer_1.__dict__["_parameters"]["weight"].data.device,
)
```

How do I get the activities with my own network class?

- **Option Default**: Use torch.nn.Sequential
- **Option A**: Write your class' forward function such that it stores the data somewhere in the class.
- **Option B**: Hooks allow to introduce custom code into the internal of PyTorch. This is a complex topic.
- **Option C**: Use a Context Manager

Option B: Hooks

```
activation = {}
```

Storage for the activities

```
def get_activation(name):
    def hook(model, input, output):
        activation[name] = output.detach()
    return hook
```

Define the hook

```
network.layer_1.register_forward_hook(get_activation("layer_1"))
output = network(fake_input)
```

Register the hook

```
for name, value in activation.items():
    print(f"Hook name: {name}")
    print(value.shape)
```

Investigate the results

Option C: Context Manager

```
from contextlib import contextmanager

@contextmanager
def collect_activations(model, layers):
    activations = {}
    handles = []
     for name, layer in layers.items():
       handles.append(
          layer.register_forward_hook(
             lambda m, i, o, name=name: activations.update({name: o.detach()})
          )
       )
     try:
       yield activations
    finally:
       for handle in handles:
          handle.remove()
```

Option C: Context Manager

```
layers_to_capture = {
    "conv1": network.layer_1,
}
with collect_activations(network, layers_to_capture) as acts:
    output = network(fake_input)

    print(acts.keys())
    print(acts["conv1"].shape)
```

Statistics Fisher Exact Test

In the case we don’t have enough computational time to compare two networks via their mean and std.

scipy.stats.fisher_exact

“Perform a Fisher exact test on a contingency table.

For a 2x2 table, the null hypothesis is that the true odds ratio of the populations underlying the observations is one, and the observations were sampled from these populations under a condition: the marginals of the resulting table must equal those of the observed table. The statistic is the unconditional maximum likelihood estimate of the odds ratio, and the p-value is the probability under the null hypothesis of obtaining a table at least as extreme as the one that was actually observed.”

```
fisher_exact(table, alternative=None, *, method=None)
```

alternative {‘two-sided’, ‘less’, ‘greater’}, optional
Defines the alternative hypothesis for 2x2 tables; unused for other table sizes. The following options are available (default is ‘two-sided’):

- ‘two-sided’: the odds ratio of the underlying population is not one
- ‘less’: the odds ratio of the underlying population is less than one
- ‘greater’: the odds ratio of the underlying population is greater than one

scipy.stats.fisher_exact

The input table is [[a, b], [c, d]].

a	b
c	d

N - a	N - b
a	b

N = Total number of test samples
a = Number of correct classification for network a
b = Number of correct classification for network b

Assumption: Both networks are tested with the same amount N of test samples

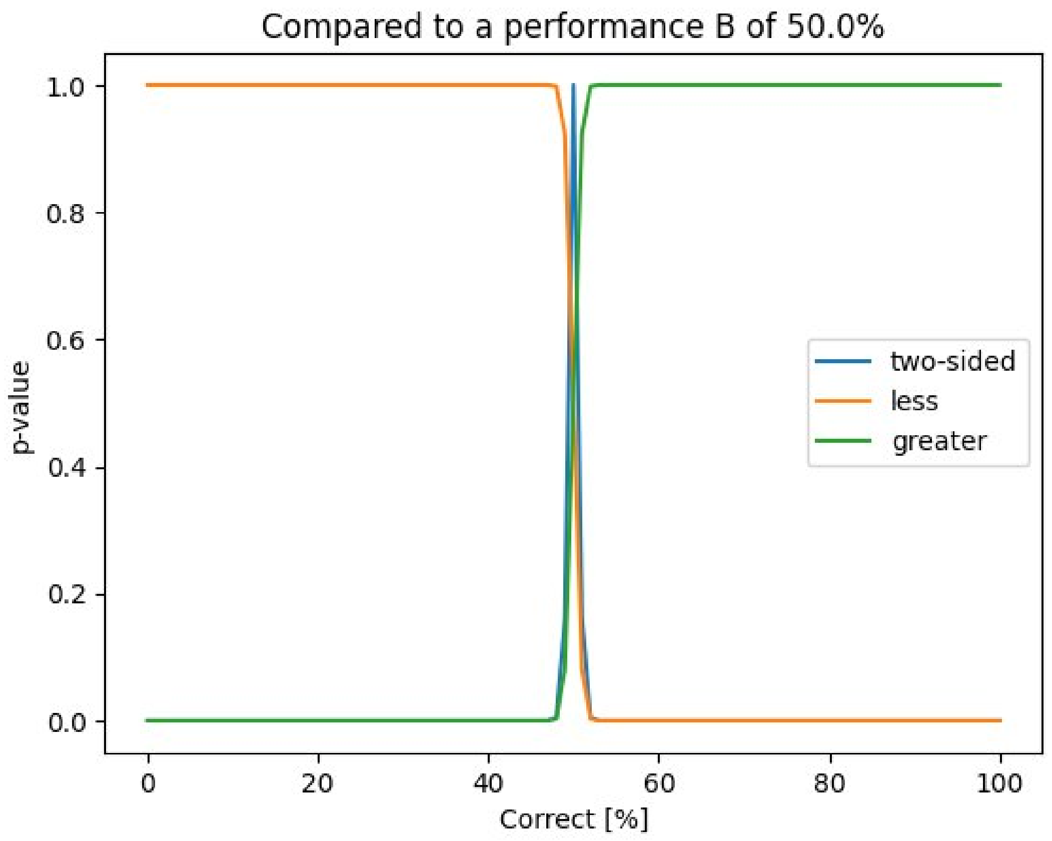
scipy.stats.fisher_exact

```
from scipy.stats
import fisher_exact
import numpy as np
import matplotlib.pyplot as plt

N: int = 10000
correct_b: int = N // 2

values = np.arange(0, N + 1, 100)
results_less = np.zeros((values.shape[0]))

for i in range(0, values.shape[0]):
    correct_a: int = int(values[i])
    res = fisher_exact(
        [[N - correct_a, N - correct_b], [correct_a, correct_b]], alternative="less"
    )
    results_less[i] = res.pvalue
```



scipy.stats.fisher_exact

```
results_greater = np.zeros((values.shape[0]))

for i in range(0, values.shape[0]):
    correct_a = int(values[i])
    res = fisher_exact(
        [[N - correct_a, N - correct_b], [correct_a, correct_b]], alternative="greater"
    )
    results_greater[i] = res.pvalue
```

scipy.stats.fisher_exact

```
results_two_sided = np.zeros((values.shape[0]))

for i in range(0, values.shape[0]):
    correct_a = int(values[i])
    res = fisher_exact(
        [[N - correct_a, N - correct_b], [correct_a, correct_b]],
        alternative="two-sided",
    )
    results_two_sided[i] = res.pvalue

plt.plot(100.0 * values / N, results_two_sided, label="two-sided")
plt.plot(100.0 * values / N, results_less, label="less")
plt.plot(100.0 * values / N, results_greater, label="greater")
plt.title(f"Compared to a performance B of {100.0 * correct_b /N}%")
plt.ylabel("p-value")
plt.xlabel("Correct [%]")
plt.legend()
plt.show()
```

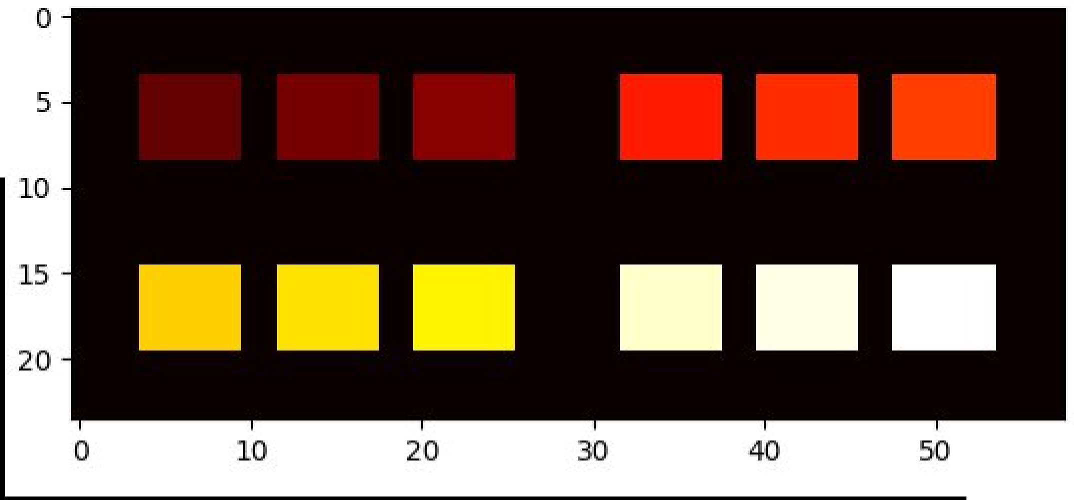
Plotting weights

make_grid

```
torchvision.utils.make_grid(tensor: Union[Tensor, List[Tensor]], nrow: int = 8,
padding: int = 2, normalize: bool = False, value_range: Optional[Tuple[int, int]] =
None, scale_each: bool = False, pad_value: float = 0.0) → Tensor
```

Make a grid of images.

make_grid



```
import torchvision as tv  # type: ignore
import torch
import matplotlib.pylab as plt

fake_weights = torch.zeros((3, 4, 5, 6))

for b in range(0, fake_weights.shape[0]):
    for c in range(0, fake_weights.shape[1]):
        fake_weights[b, c, ...] = 5 + b + c * 10

grid_image = tv.utils.make_grid(fake_weights, nrow=fake_weights.shape[0])
print(grid_image.shape) # torch.Size([4, 9, 26])
grid_image = tv.utils.make_grid(grid_image.unsqueeze(1), nrow=2)[0, ...]
print(grid_image.shape) # torch.Size([24, 58])
plt.imshow(grid_image, cmap="hot")
plt.show()
```

If dim channel = 1 goes in
dim channel = 3 comes out