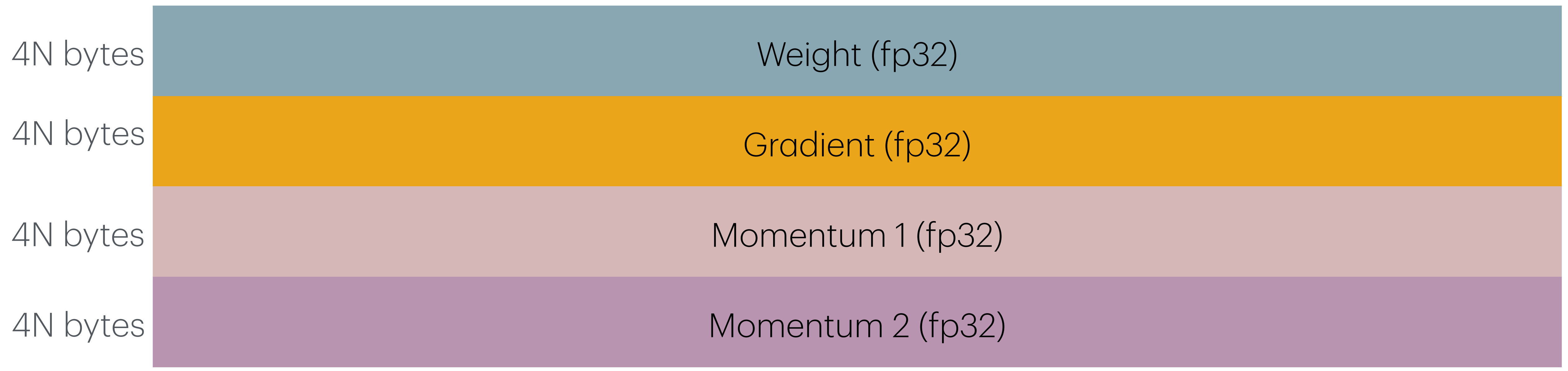
# Checkpointing

Philipp Krähenbühl, UT Austin

# Training large models

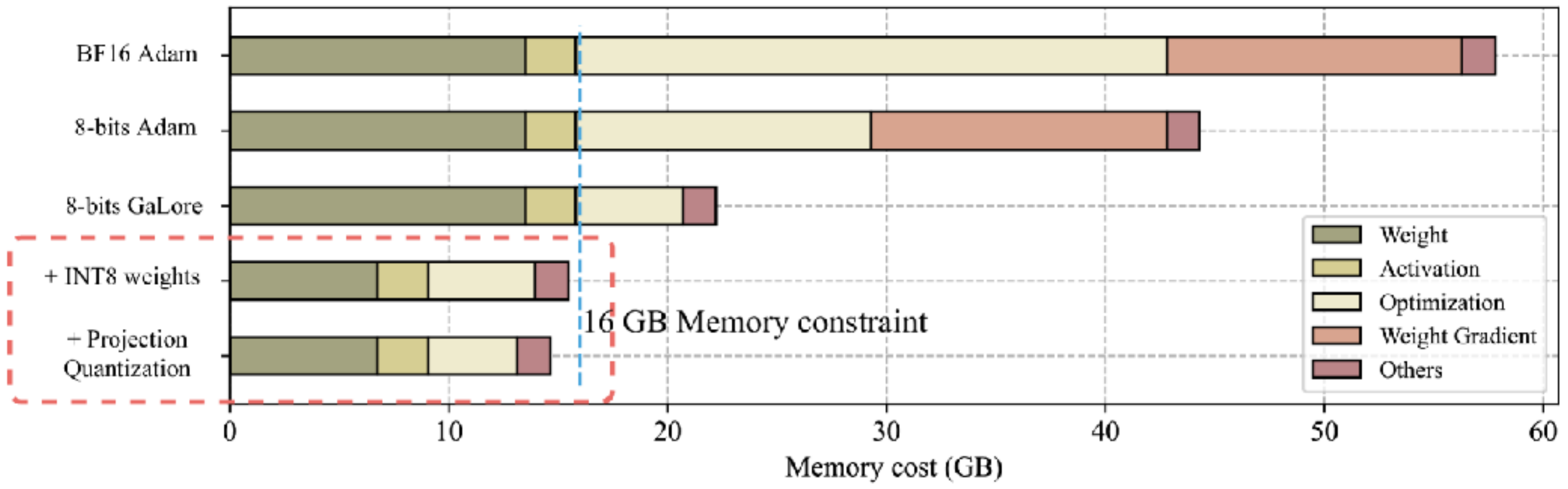
## Memory requirements

- Without optimization:
  - Model parameters: N
  - Weights: N floats
  - Gradients: N floats
  - Momentum: N floats
  - 2nd momentum (ADAM): N floats
- 16N bytes without counting activations

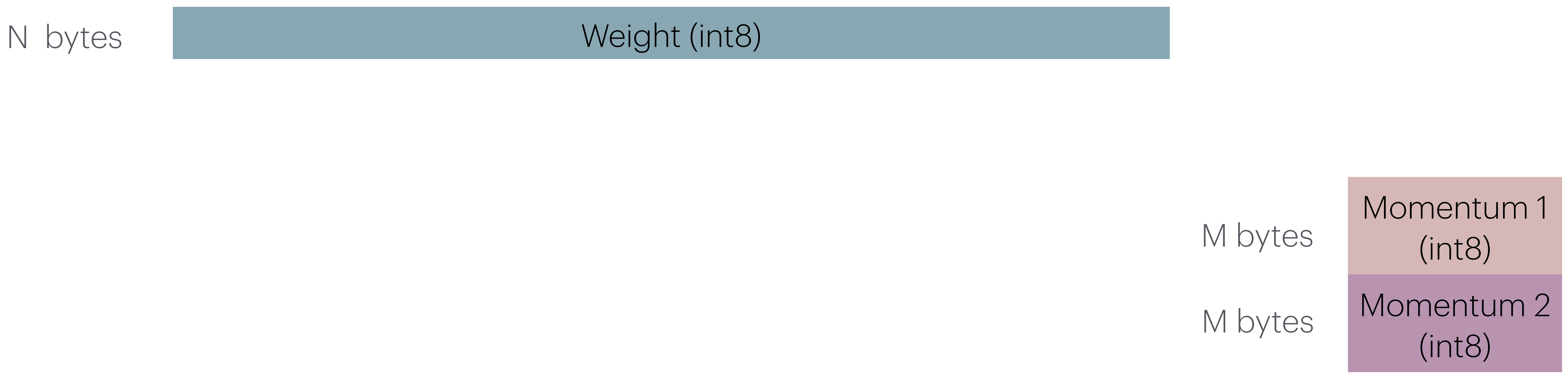


# Training Q-Galore models

## Memory requirements




- Q-Galore
  - Model parameters: N, Galore parameters M
  - Weights: N int8
  - Momentum: M int8
  - 2nd momentum (ADAM): M int8
  - Projection: int4
- N+2M + |projection| bytes without activations

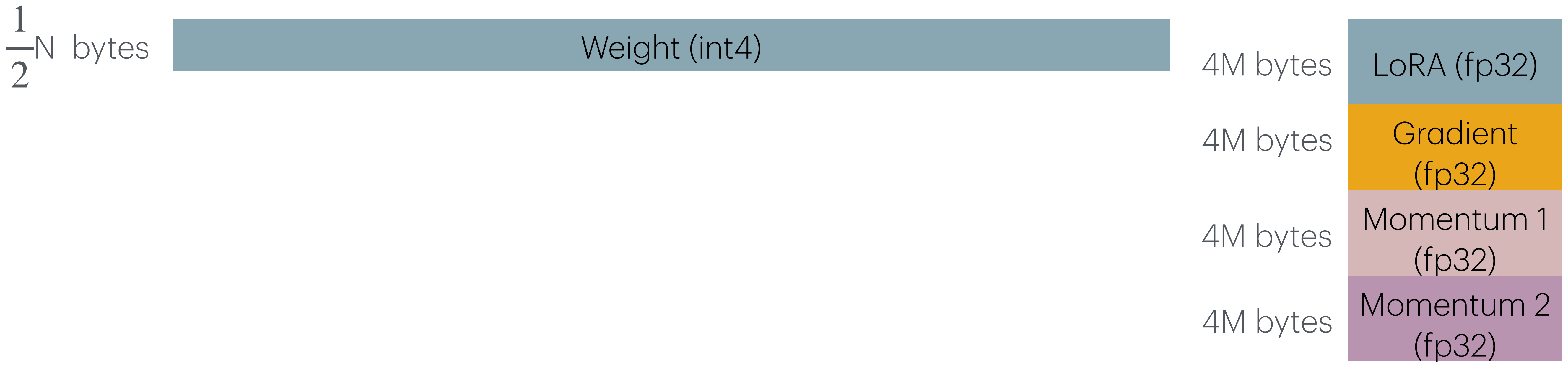



[1] Zhenyu Zhang, et al. Q-GaLore: Quantized GaLore with INT4 Projection and Layer-Adaptive Low-Rank Gradients. 2024

# Training QLoRA models

## Memory requirements

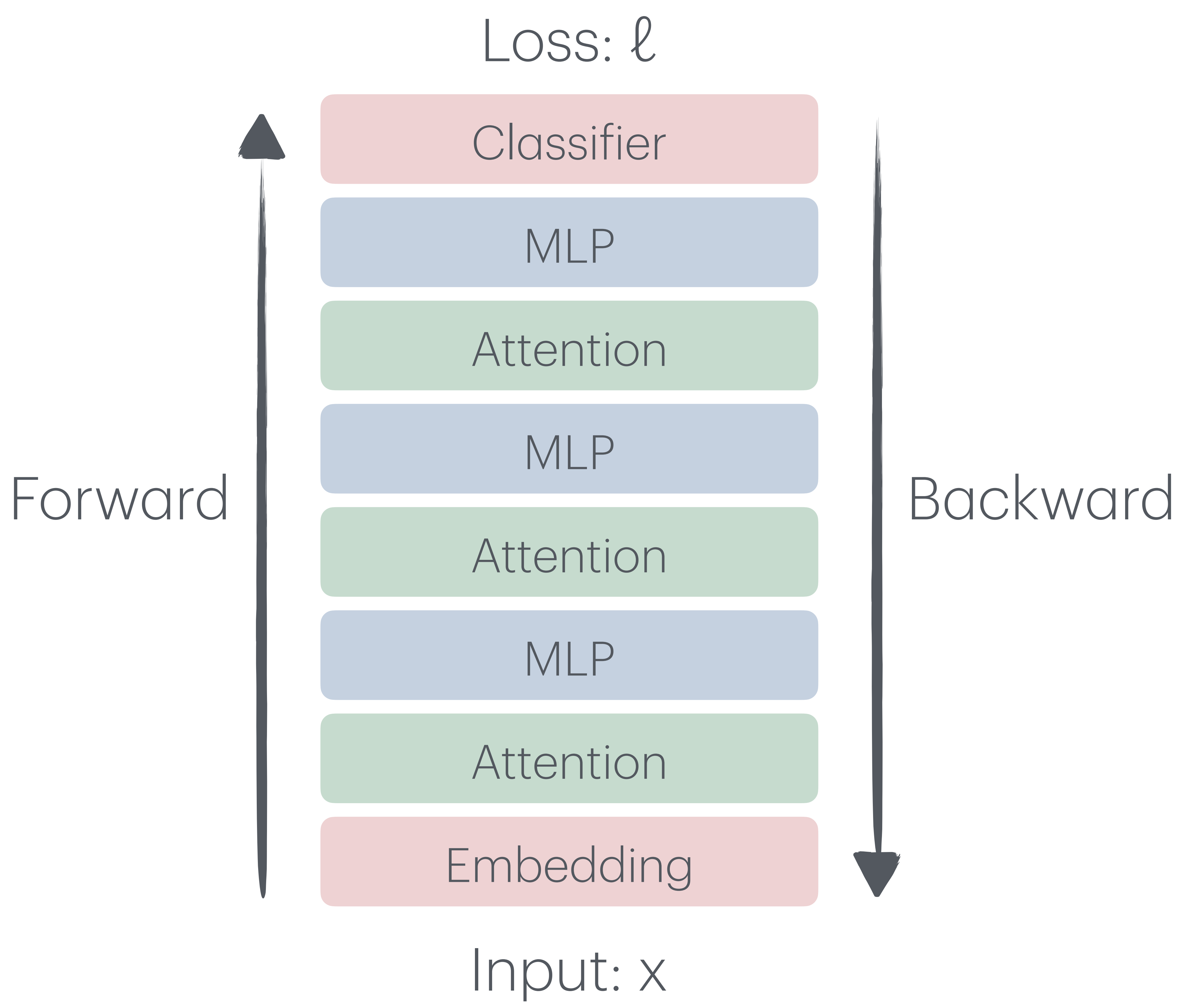
- QLoRA
  - Model parameters: N, LoRA param M
  - Weights: N int4, M floats
  - Gradients: M floats
  - Momentum: M floats
  - 2nd momentum (ADAM): M floats
- $\frac{1}{2}$N+16M bytes without activations
- M often ~1-5% of N



# Backpropagation

## A closer look

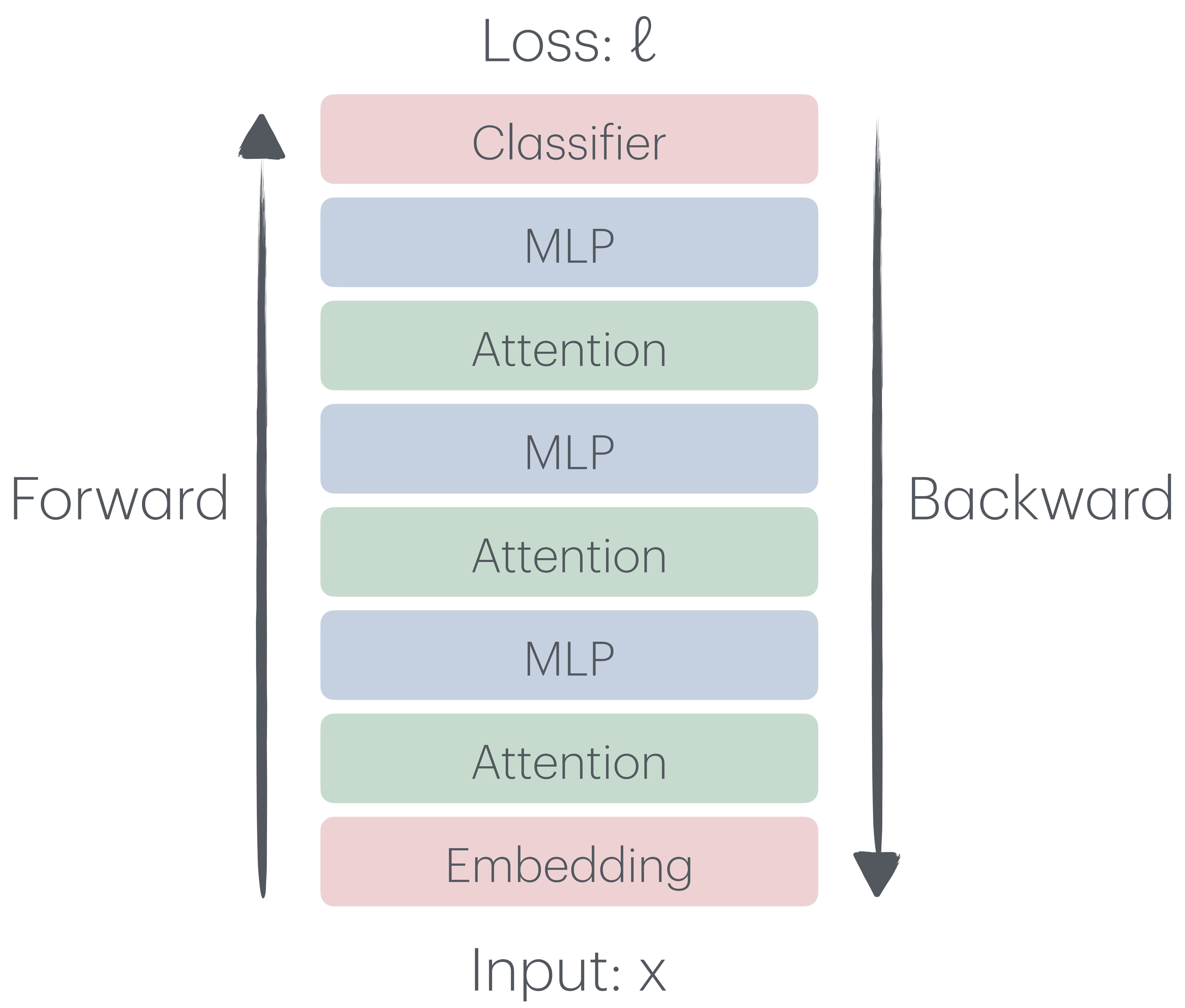
- Linear layer $y = W^\top x$
  - Gradient: $\frac{\partial}{\partial W} y = x y^\top$
  - Backprop: $\frac{\partial}{\partial x} y = W^\top$
- Nonlinear layer $y = f(x)$
  - Backprop: $\frac{\partial}{\partial x} y = \nabla f(x)$



# Backpropagation

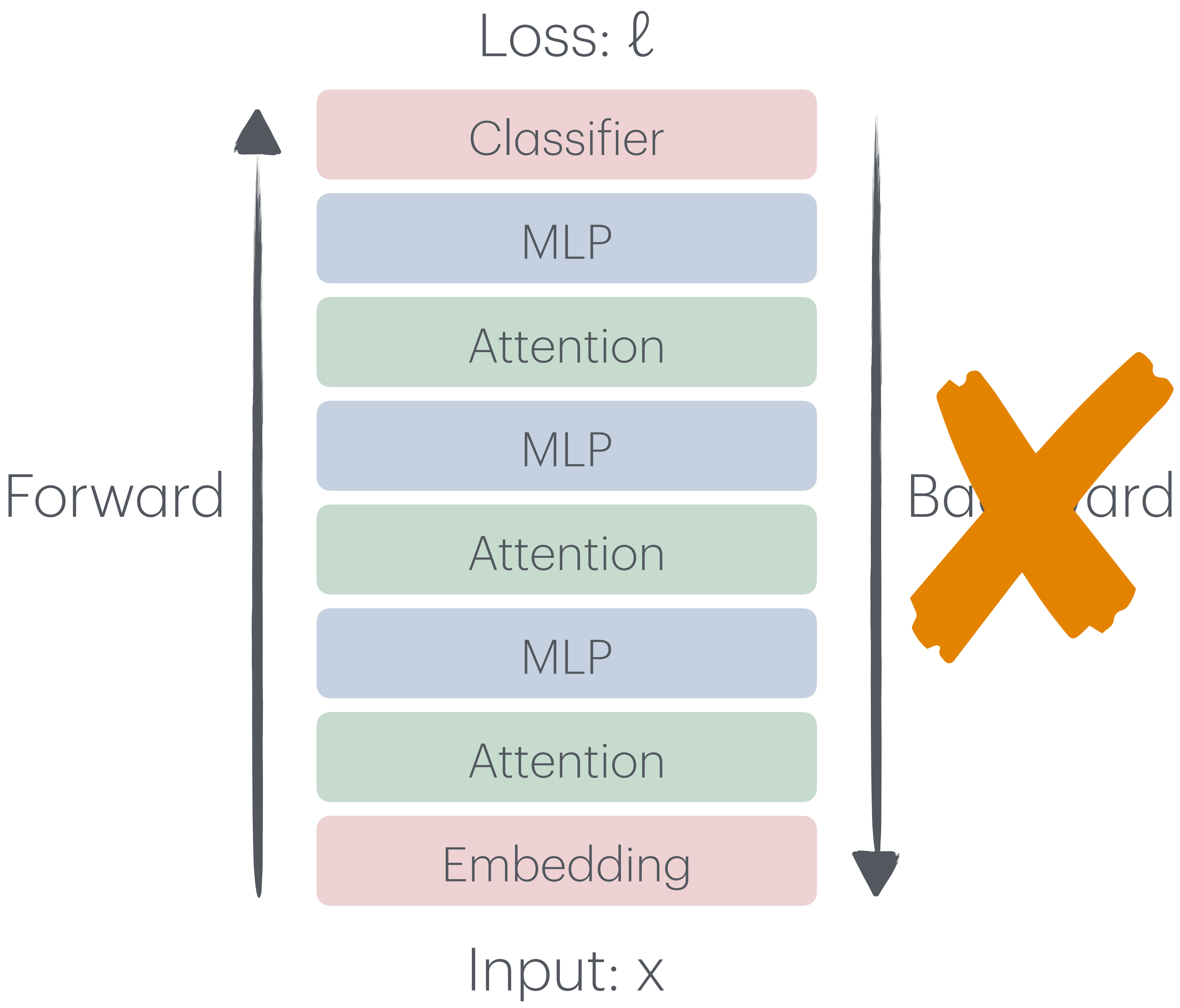
## A closer look

- Forward
  - Store activation
    - Backprop of non-linear layers
    - Weight gradient of linear layers
- Backward
  - Compute gradient (allocate memory)
  - Discard activation (free memory)



# Backpropagation

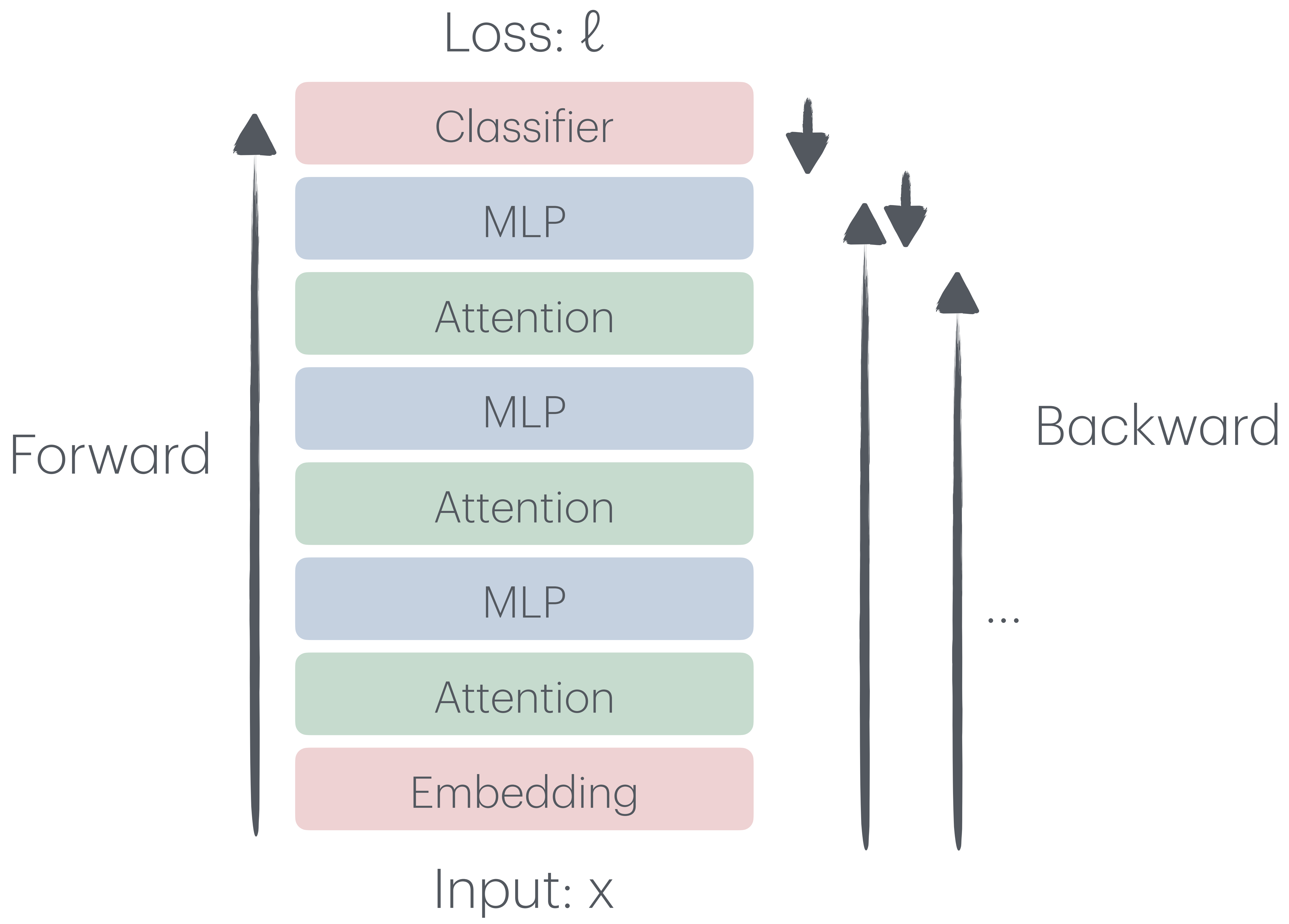
- Forward only
  - Memory efficient
  - Reuse memory buffers
  - `with torch.no_grad():`



# Backpropagation

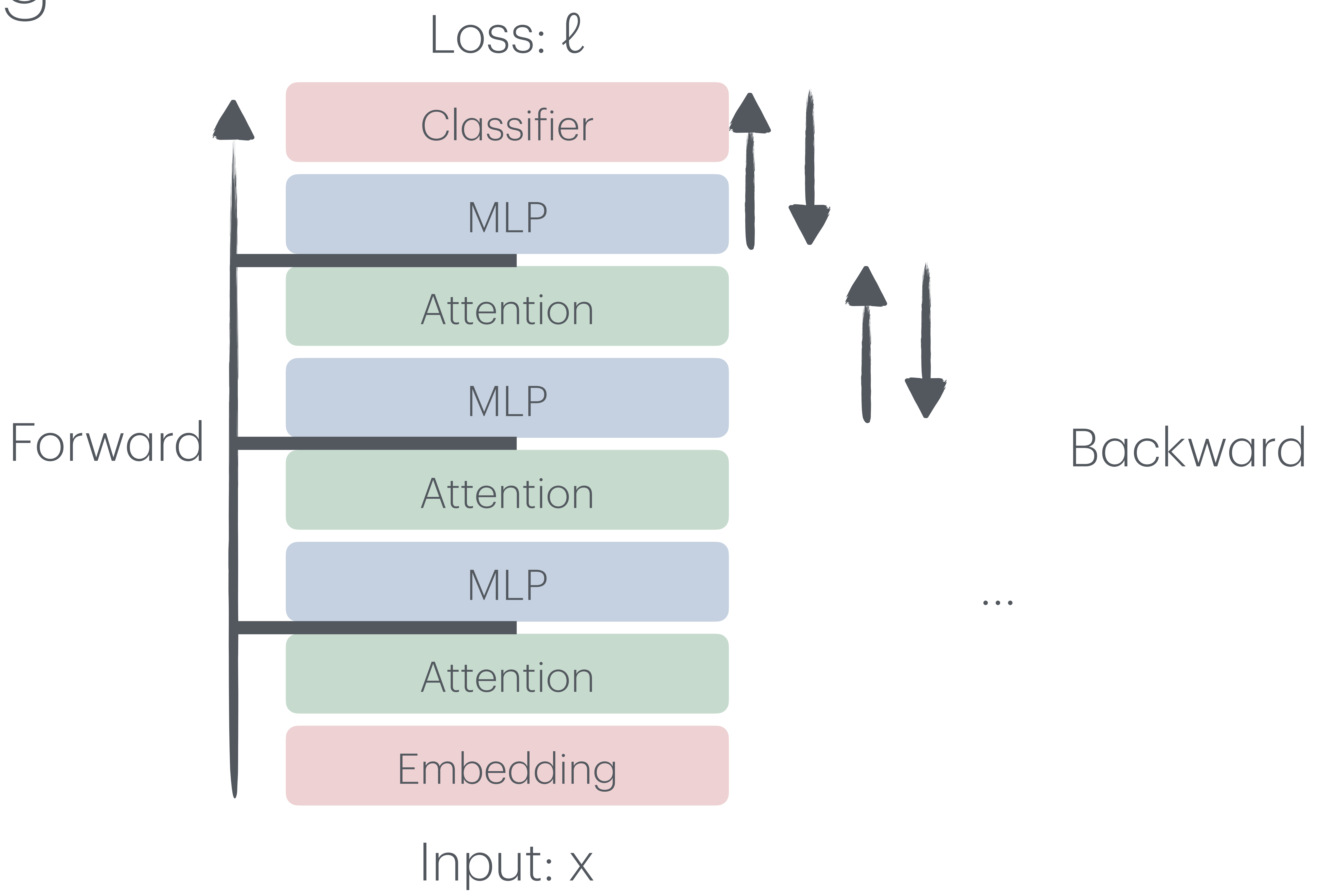
## Without storing activations

- Forward
  - `with torch.no_grad():`
- Backward
  - **Recompute activation**
  - Compute gradient (allocate memory)
  - Discard activation (free memory)
- No additional memory!
- Very slow D forward passes for one backward



# Activation checkpointing

- Forward
  - `with torch.no_grad():`
    - In blocks
- Backward
  - **Recompute activation**
    - Within block
  - Compute gradient (allocate memory)
  - Discard activation (free memory)
- Ideally sqrt(D) less memory
- 2x forward passes for one backward




[1] Tianqi Chen, et al. Training deep nets with sublinear memory cost. 2016

# Activation checkpointing in practice

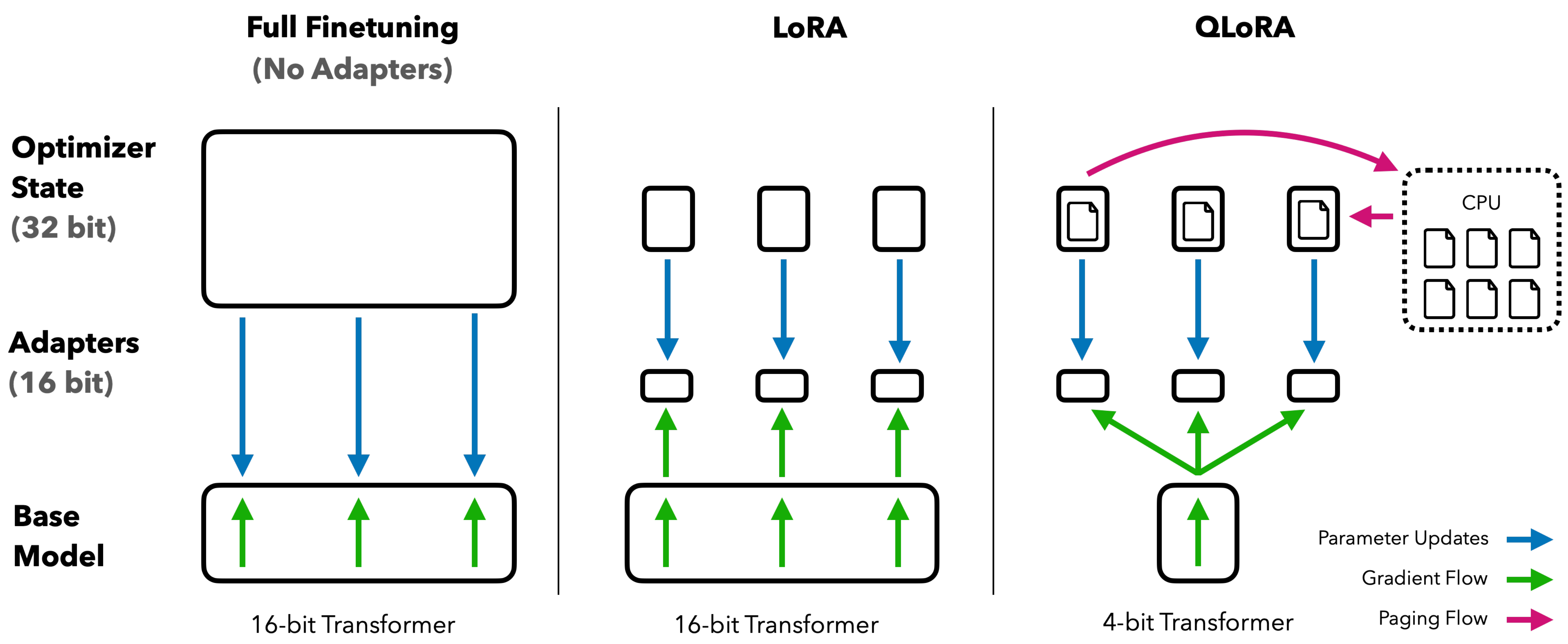
- Practical considerations
  - Need to control randomness
    - First and second forward should match
  - Need to wrap model

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable, Function
import torch.utils.checkpoint as checkpoint

class ConvBNReLU(nn.Module):

    def __init__(self, in_planes, out_planes):

        super(ConvBNReLU, self).__init__()
        self.conv1 = nn.Conv2d(in_planes, out_planes, kernel_size=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_planes)
        self.relu1 = nn.ReLU(inplace=True)

    def forward(self, x):
        out = self.relu1(self.bn1(self.conv1(x)))
        return out

class DummyNet(nn.Module):
    def __init__(self):
        super(DummyNet, self).__init__()
        self.features = nn.Sequential(OrderedDict([
            ('conv1', nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1, bias=False)),
            ('bn1', nn.BatchNorm2d(16)),
            ('relu1', nn.ReLU(inplace=True)),
        ]))

        # The module that we want to checkpoint
        self.module = ConvBNReLU(16, 64)

        self.final_module = ConvBNReLU(64, 64)

    def forward(self, x):
        out = self.features(x)
        out = checkpoint.checkpoint(self.module, out)
        out = self.final_module(out)
        return out
```

[1] Tianqi Chen, et al. Training deep nets with sublinear memory cost. 2016
[2] Priya Goyal, https://github.com/prigoyal/pytorch_memonger

# Offloading

- Certain inputs (i.e. text) have variable length
  - Variable memory use for activation checkpoints
  - “Unlucky” batches will blow up GPU memory
- Solution
  - Offload to CPU if needed
  - CUDA does this automatically with unified memory architecture (tricky in PyTorch)

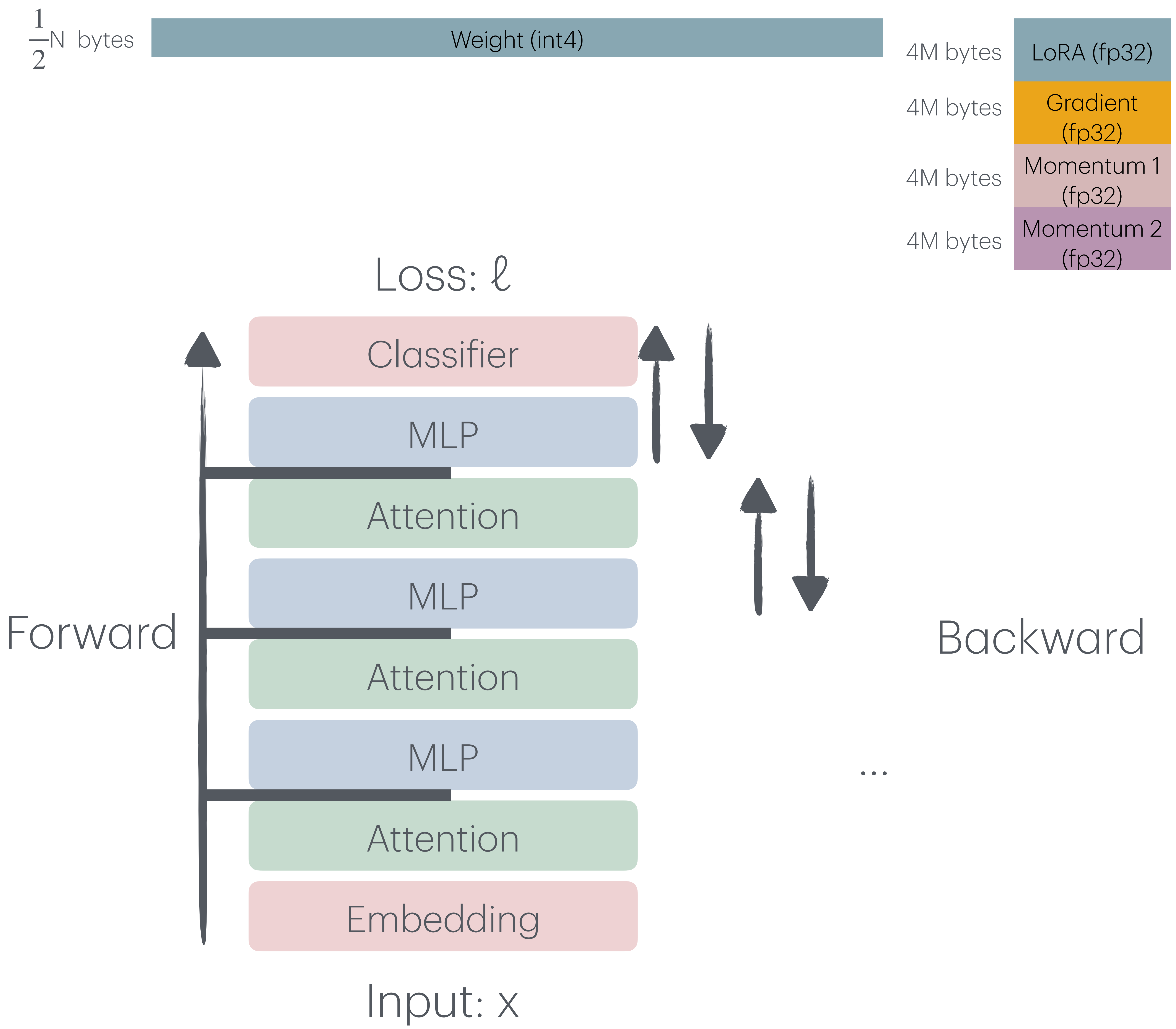



[1] Zhenyu Zhang, et al. Q-GaLore: Quantized GaLore with INT4 Projection and Layer-Adaptive Low-Rank Gradients. 2024

# Memory efficient model training

## Memory requirements

- QLoRA
  - Model parameters: N, LoRA param M
  - Weights: N int4, M floats
  - Gradients: M floats
  - Momentum: M floats
  - 2nd momentum (ADAM): M floats
- $\frac{1}{2}$N+16M bytes; M often ~1-5% of N
- $O(\sqrt{D})$ gradient checkpoint; two forward passes



# References


- [1] Zhenyu Zhang, et al. Q-GaLore: Quantized GaLore with INT4 Projection and Layer-Adaptive Low-Rank Gradients. 2024. (link)
- [2] Tianqi Chen, et al. Training deep nets with sublinear memory cost. 2016. (link)
- [3] Priya Goyal, https://github.com/prigoyal/pytorch_memonger