Introduction to PyTorch

and Scaling PyTorch Code Using LightningLite

Sebastian Raschka & Adrian Wälchli

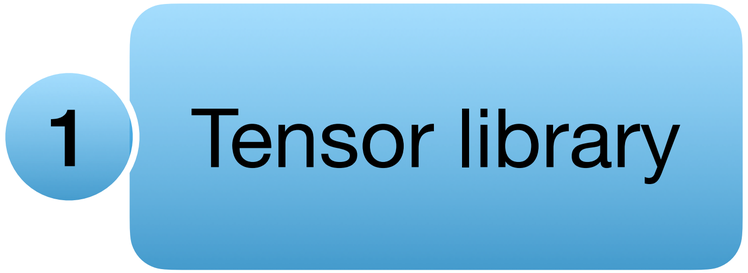
Data Umbrella

May 10th, 2022

What is PyTorch?

Tensor library

What is PyTorch?



What is PyTorch?

1 Tensor library

2 Automatic differentiation engine

What is PyTorch?

1 Tensor library

2 Automatic differentiation engine

What is PyTorch?

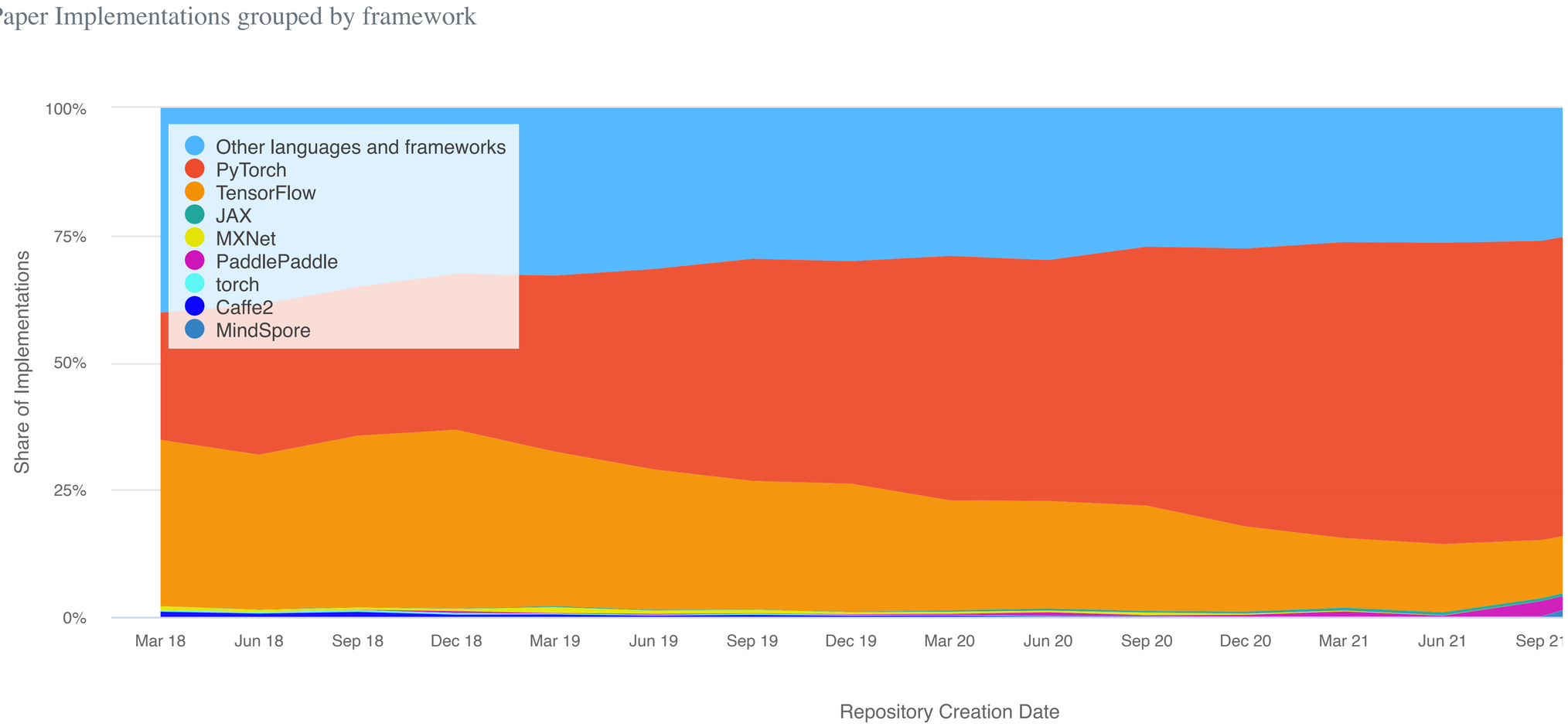
3 Deep learning library

Sounds like TensorFlow?

Sounds like TensorFlow?

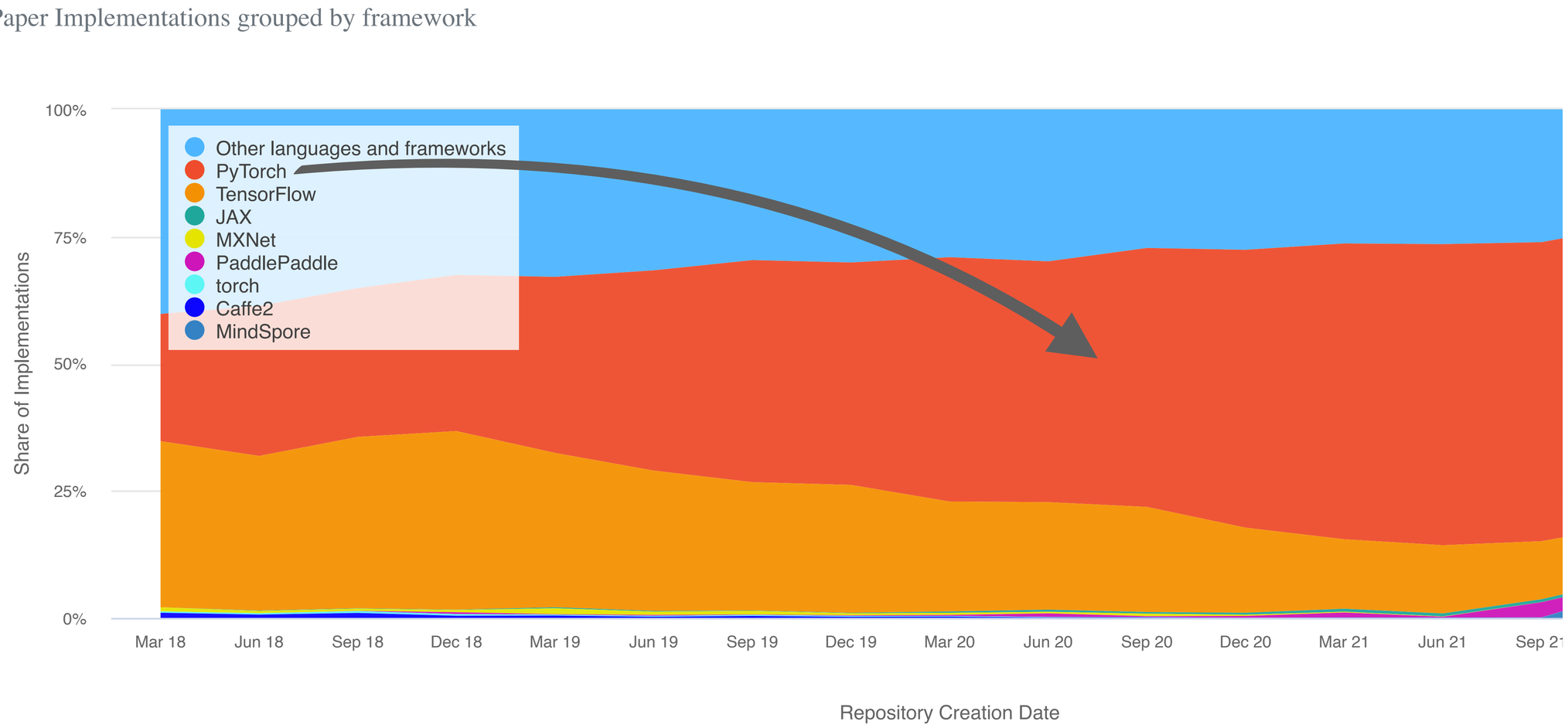
Yes, kind of.

There is probably a reason why so many people like PyTorch …



https://paperswithcode.com/trends

There is probably a reason why so many people like PyTorch …

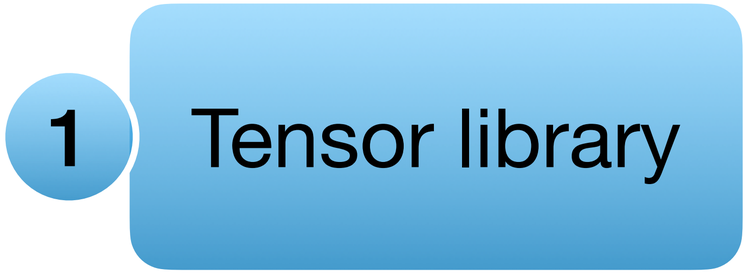


https://paperswithcode.com/trends

As a former TF user, why do I like PyTorch?

As a former TF user, why do I like PyTorch?

Stay tuned!



What is PyTorch?

Tensor library

Scalar
(rank-0 tensor)

```
import torch

a = torch.tensor(1.)
a.shape
```

```
torch.Size([])
```

Tensor library

Scalar (rank-0 tensor)

```
import torch

a = torch.tensor(1.)
a.shape
```

```
torch.Size([])
```

Vector (rank-1 tensor)

```
a = torch.tensor([1., 2., 3.])
a.shape
```

```
torch.Size([3])
```

Tensor library

Scalar (rank-0 tensor)

```
import torch

a = torch.tensor(1.)
a.shape
```

```
torch.Size([])
```

Vector (rank-1 tensor)

```
a = torch.tensor([1., 2., 3.])
a.shape
```

```
torch.Size([3])
```

Matrix (rank-2 tensor)

```
a = torch.tensor([[1., 2., 3.],
                  [2., 3., 4.]])
a.shape
```

```
torch.Size([2, 3])
```

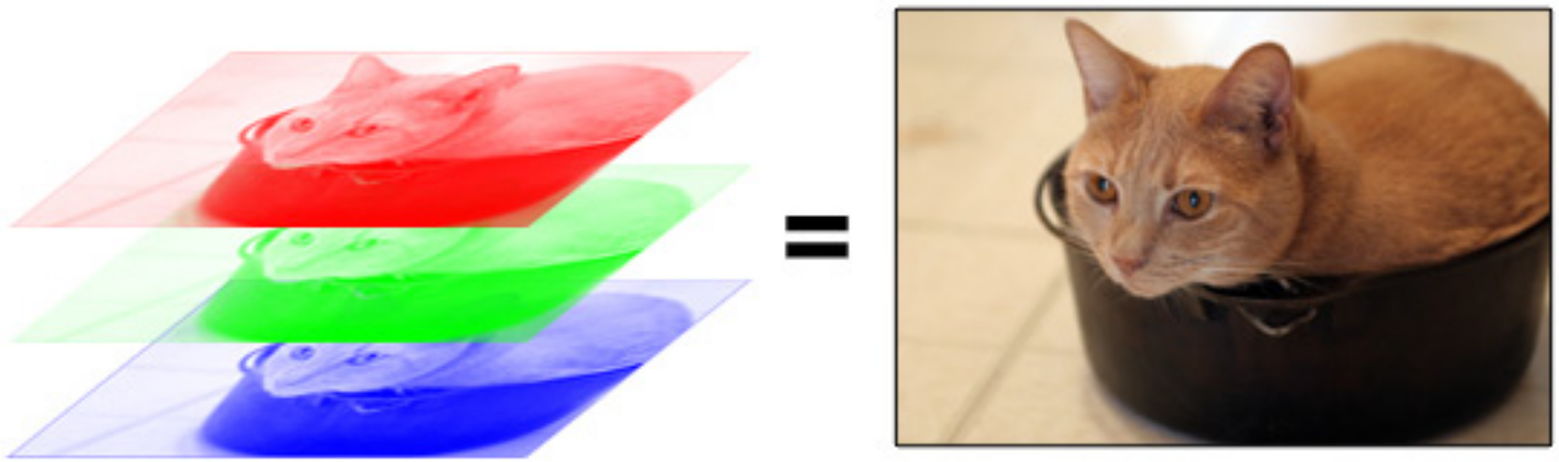
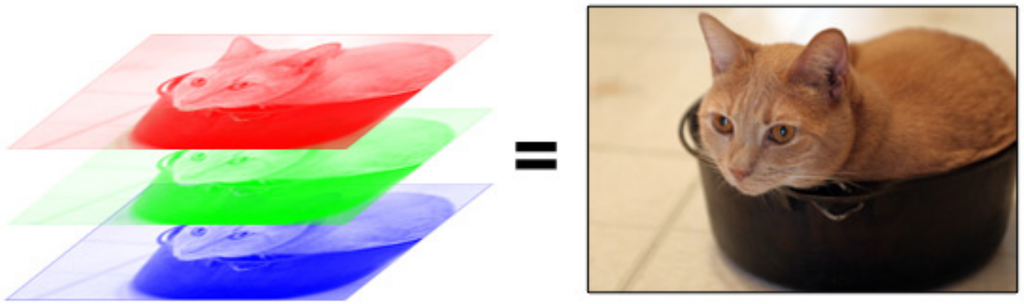


Image Source: https://code.tutsplus.com/tutorials/create-a-retro-crt-distortion-effect-using-rgb-shifting--active-3359

Color image as a stack of matrices



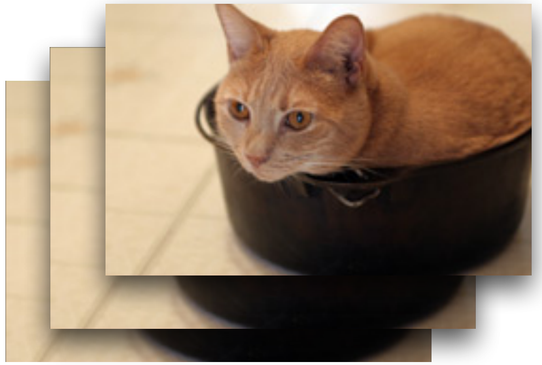
Color image as a stack of matrices

3D tensor
(rank-3 tensor)

```
a = torch.tensor([[[1., 2., 3.],
                   [2., 3., 4.]],
                  [[5., 6., 7.],
                   [8., 9., 10.]]])
a.shape
```

```
torch.Size([2, 2, 3])
```

A stack of multiple color images



A stack of multiple color images

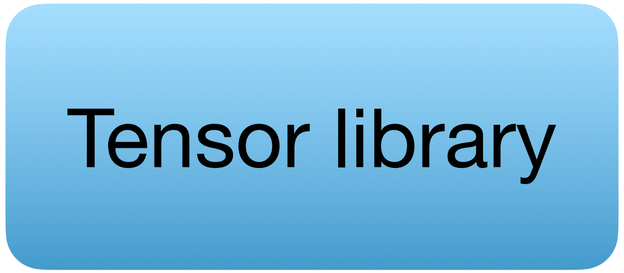
4D tensor
(rank-4 tensor)

```
b = torch.stack((a, a))
b.shape
```

```
torch.Size([2, 2, 2, 3])
```

```
b
```

```
tensor([[[[ 1.,  2.,  3.],
          [ 2.,  3.,  4.]],

         [[ 5.,  6.,  7.],
          [ 8.,  9., 10.]]],


        [[[ 1.,  2.,  3.],
          [ 2.,  3.,  4.]],

         [[ 5.,  6.,  7.],
          [ 8.,  9., 10.]]]])
```



Tensor library == array library

Tensor library

torch.tensor ≈ numpy.array

Tensor library

torch.tensor ≈ numpy.array

+ GPU support `a.to('cuda')`

Tensor library

torch.tensor ≈ numpy.array

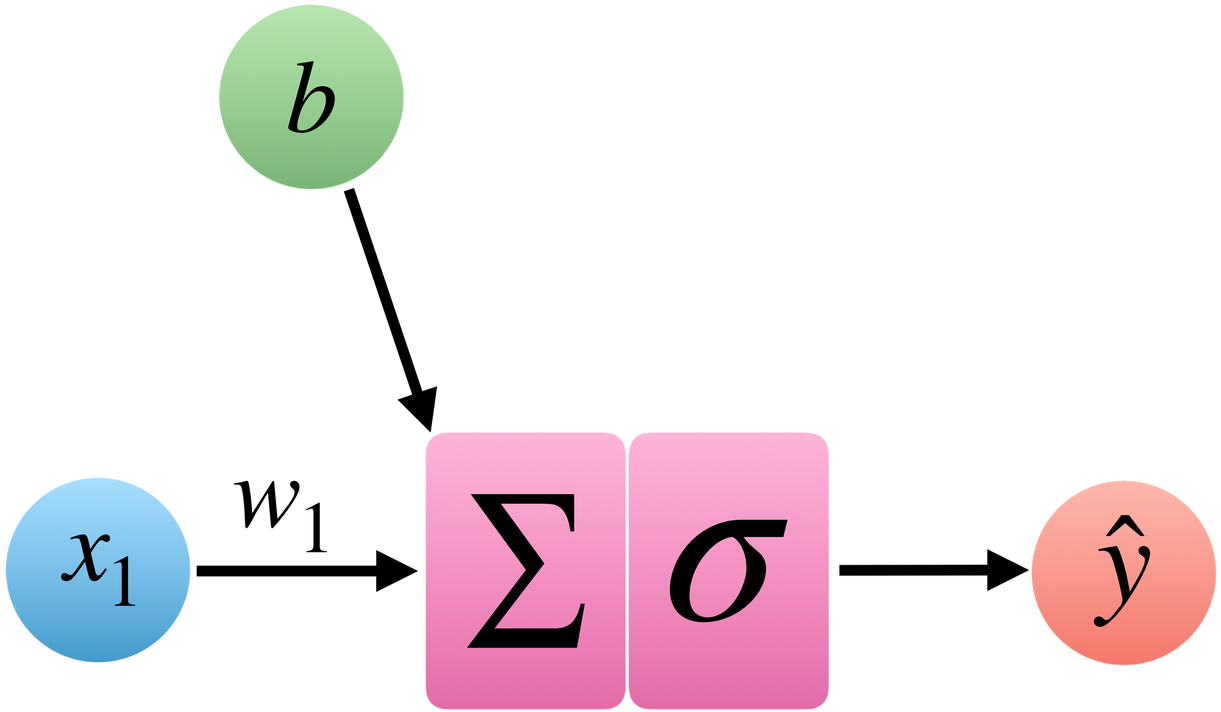
+ GPU support `a.to('cuda')`

+ automatic differentiation support

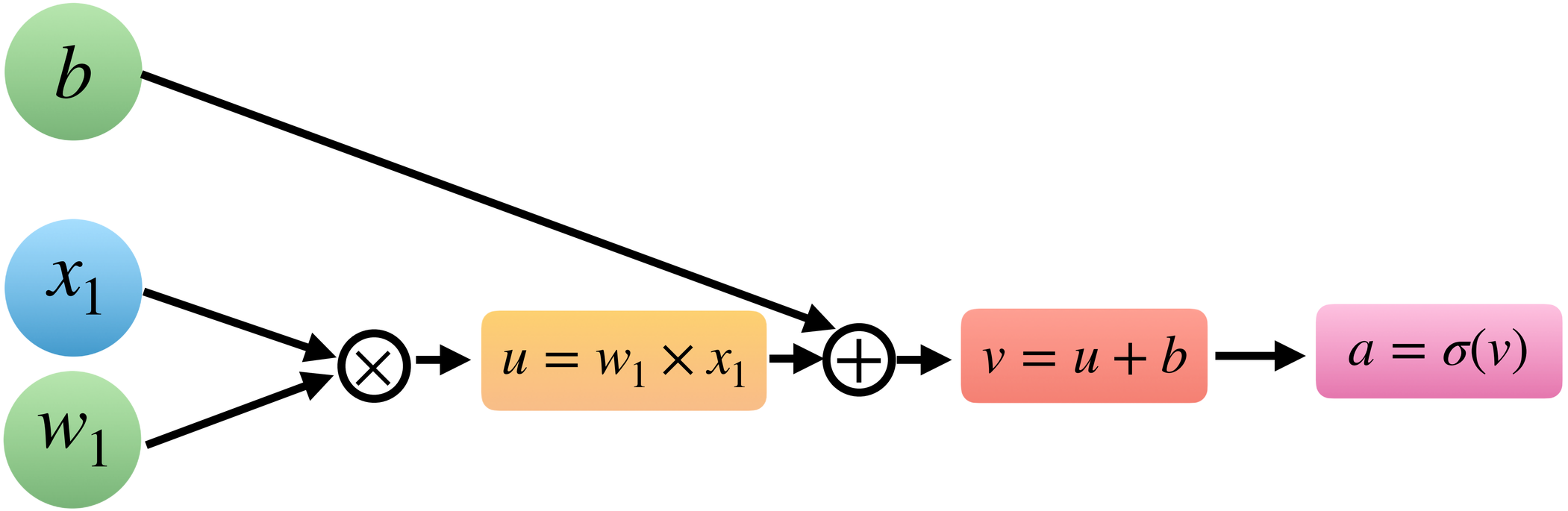
2 Automatic differentiation engine

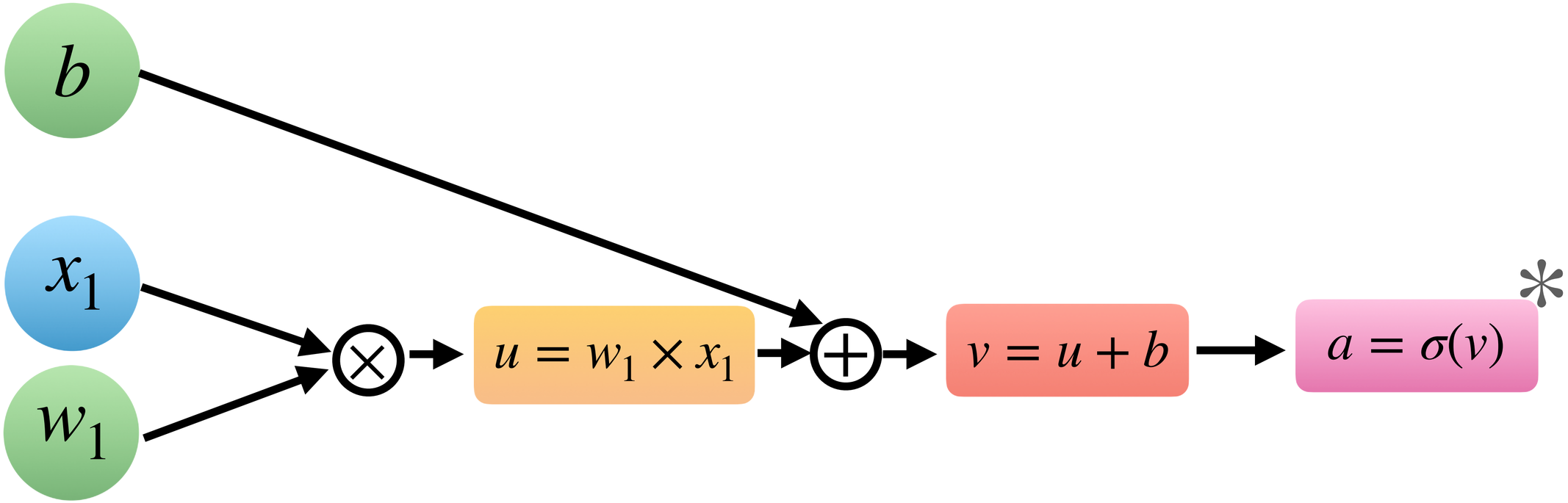
What is PyTorch?

A logistic regression model



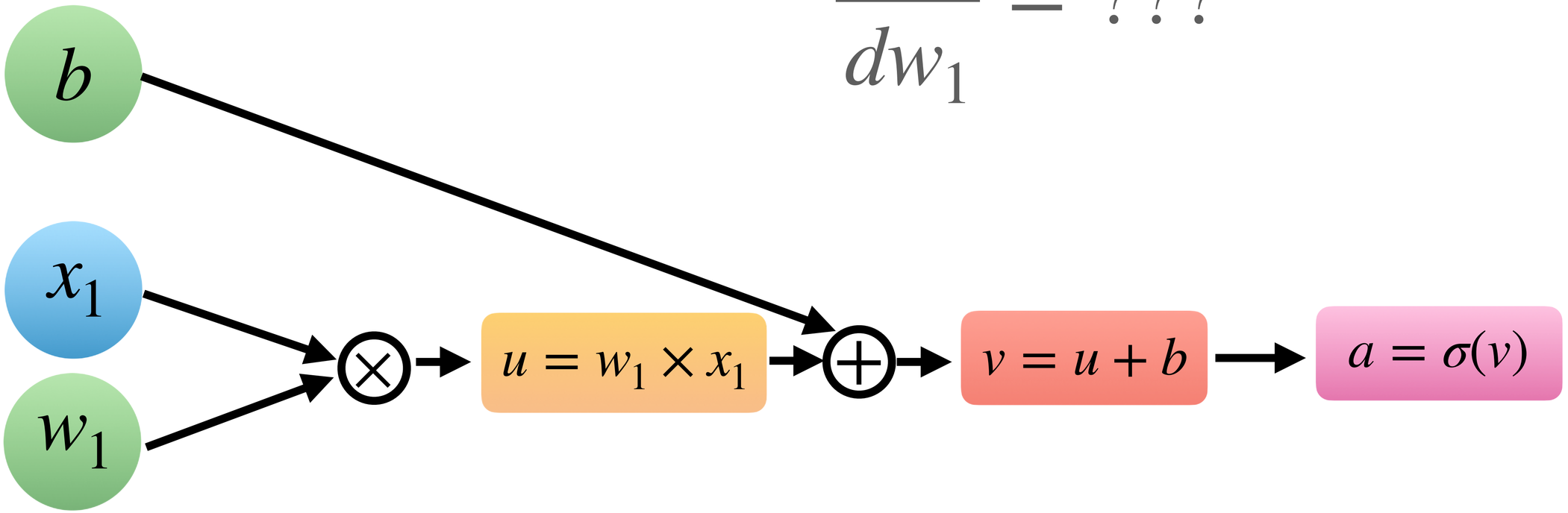
A computation graph



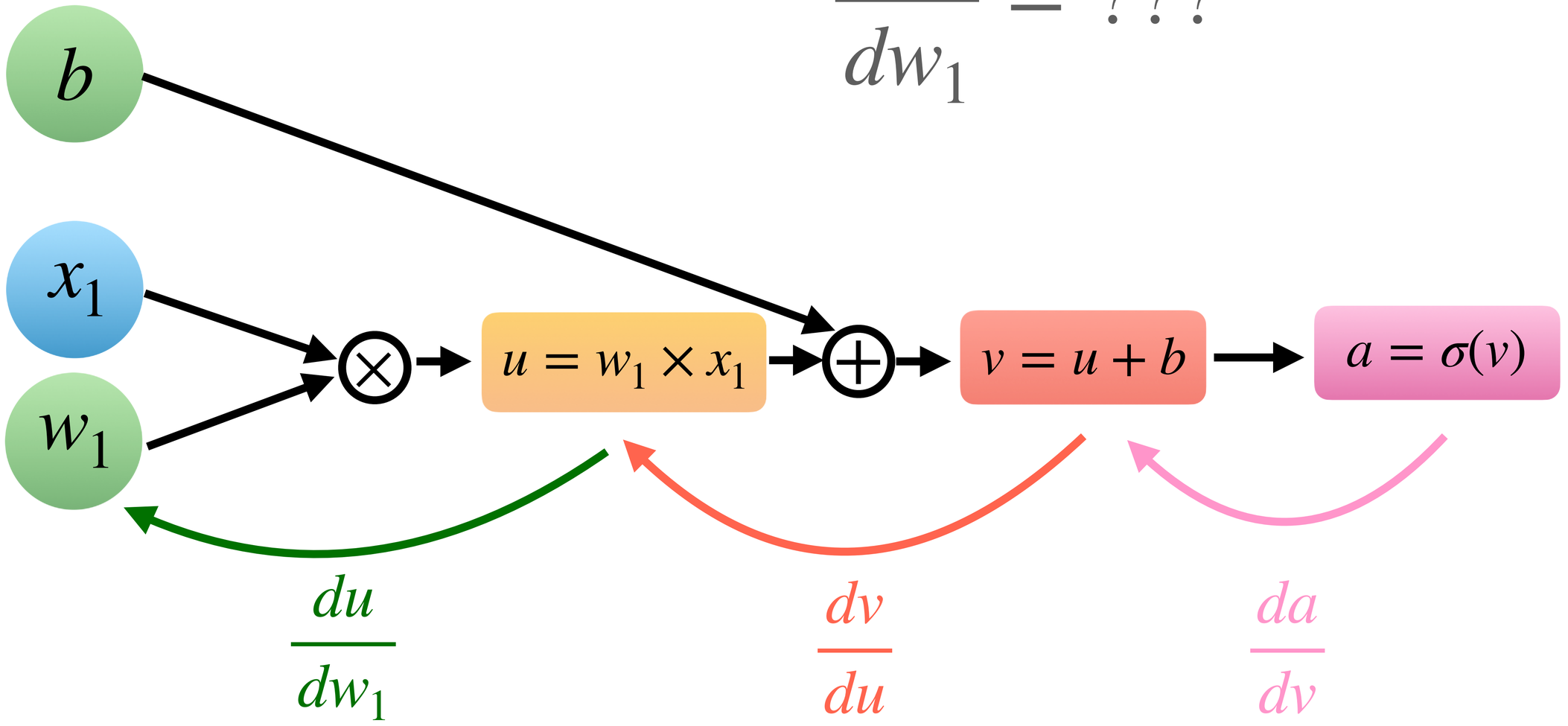


* $\sigma(z) = \dfrac{1}{1 + e^{-z}}$

$$\frac{da}{dw_1} = ???$$

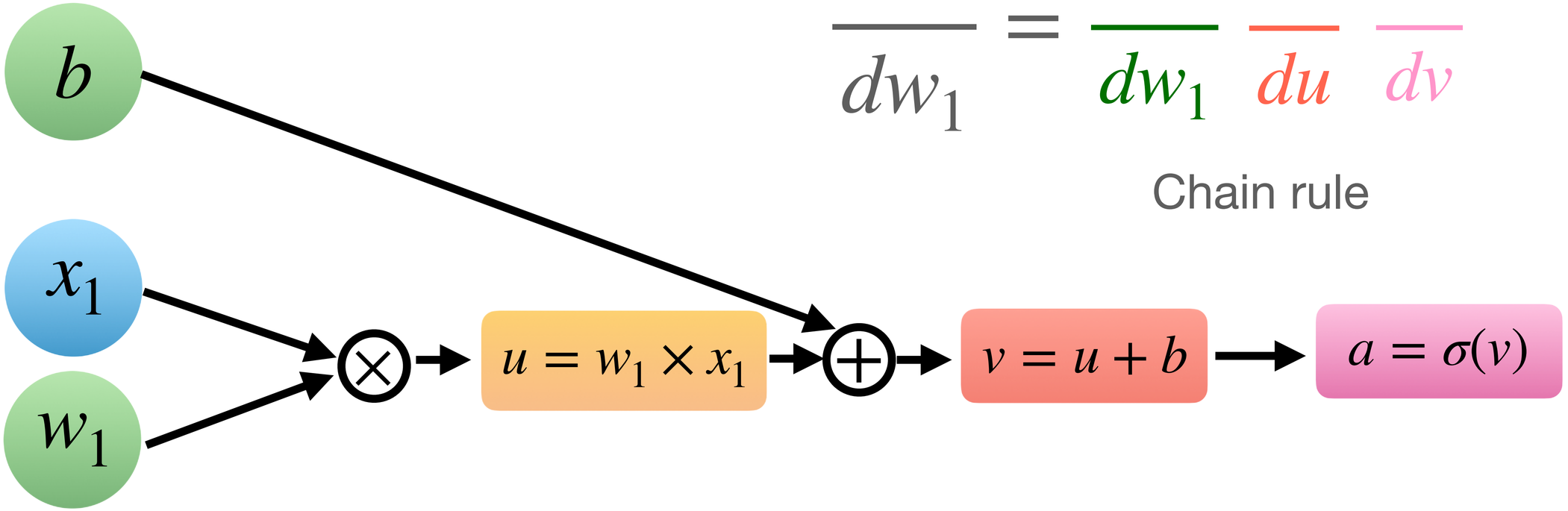


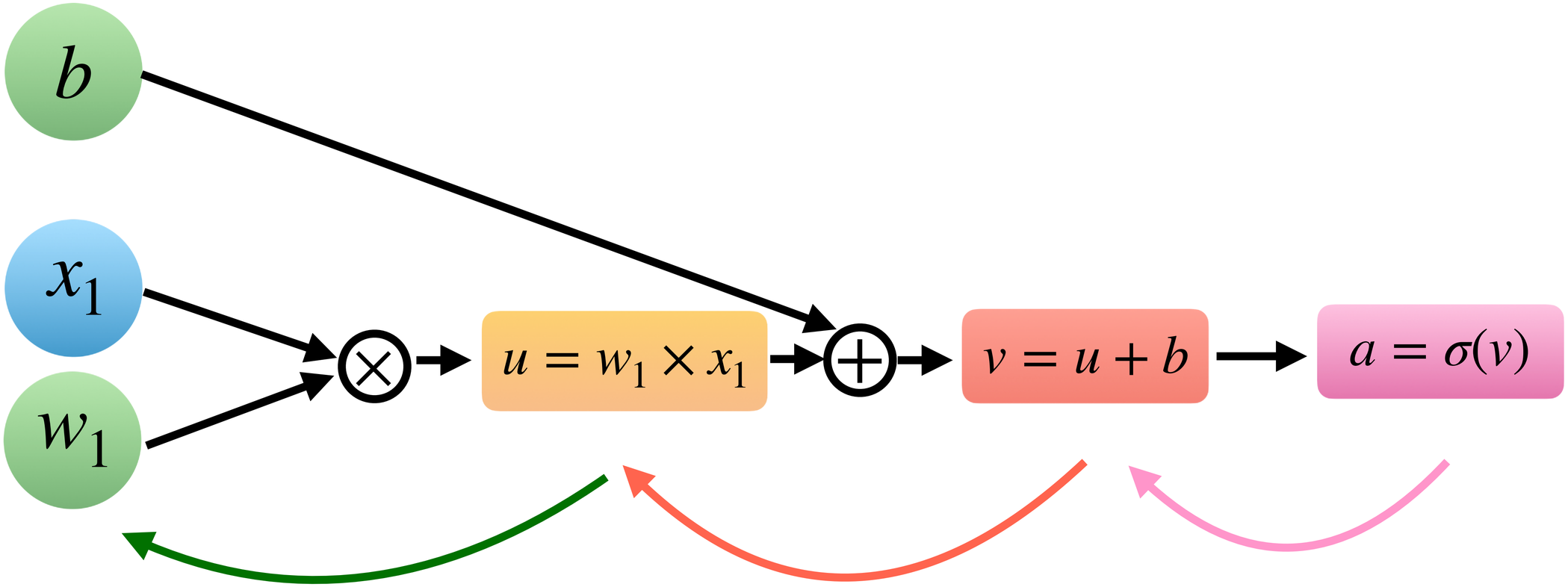
$$\frac{da}{dw_1} = ???$$



$$\frac{da}{dw_1} = \frac{du}{dw_1}\frac{dv}{du}\frac{da}{dv}$$

Chain rule

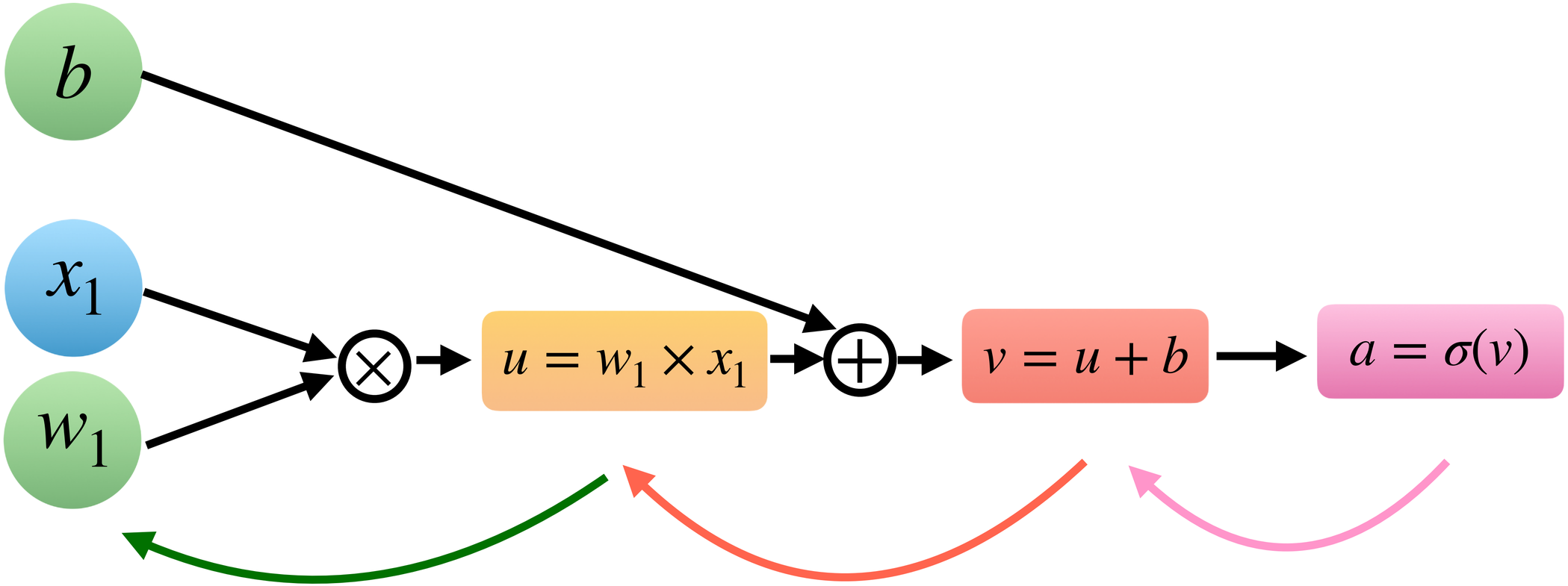




Derive terms individually

$$\frac{du}{dw_1} \qquad \frac{dv}{du} \qquad \frac{da}{dv}$$

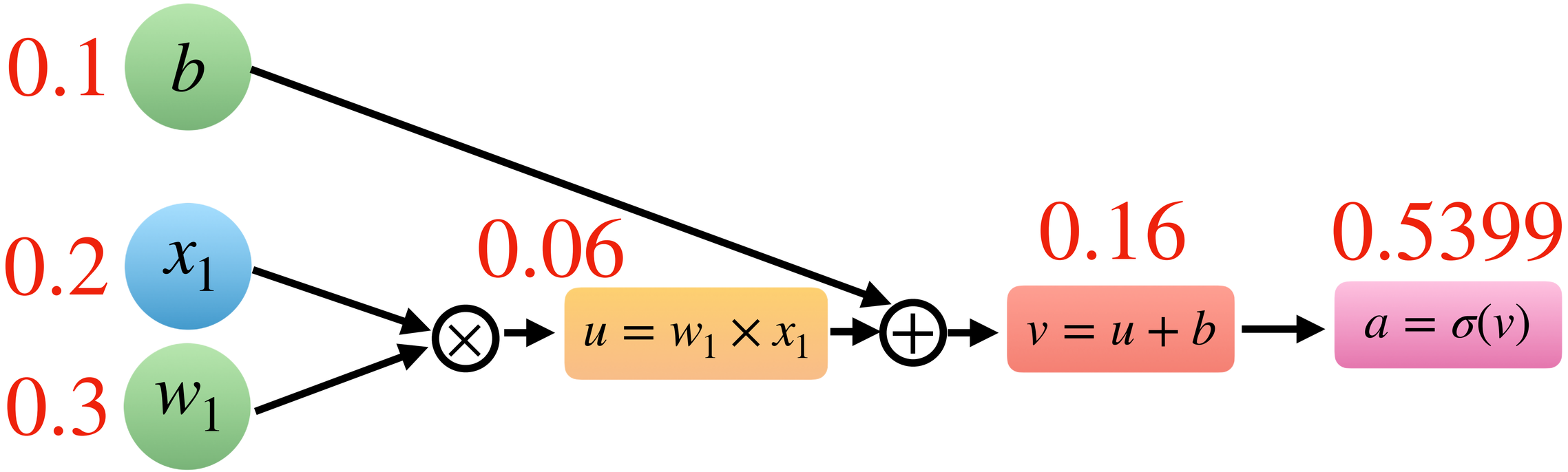
$$x_1 \qquad 1 \qquad \sigma(v)(1 - \sigma(v))$$

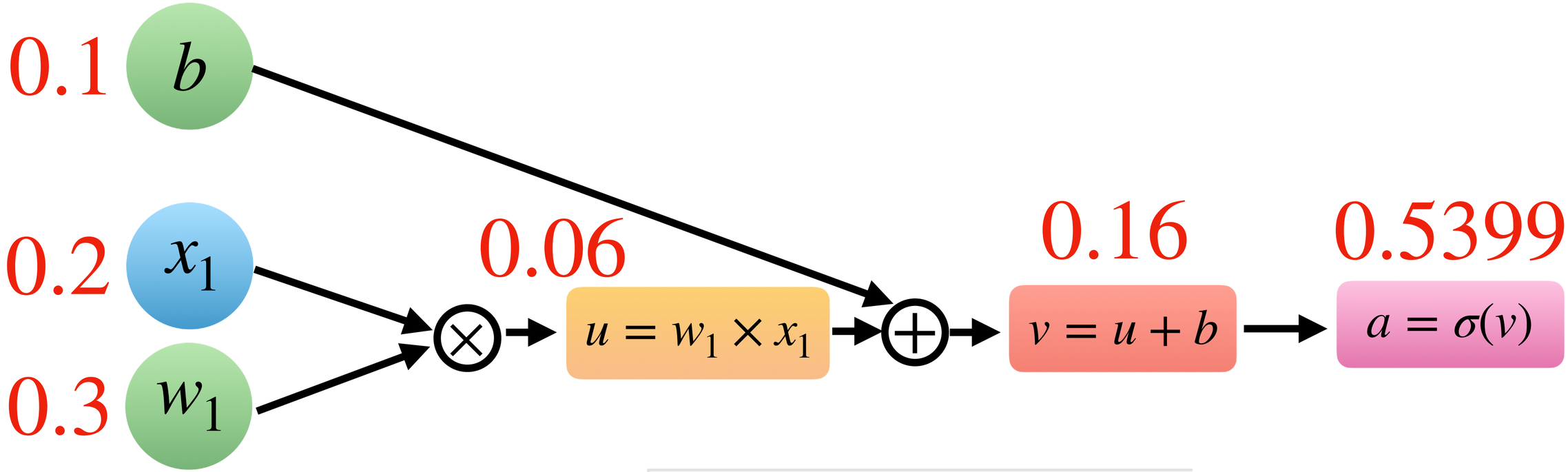


And then combine

$$\frac{du}{dw_1} \qquad \frac{dv}{du} \qquad \frac{da}{dv}$$

$$\frac{da}{dw_1} = x_1 \times 1 \times \sigma(v)(1 - \sigma(v))$$

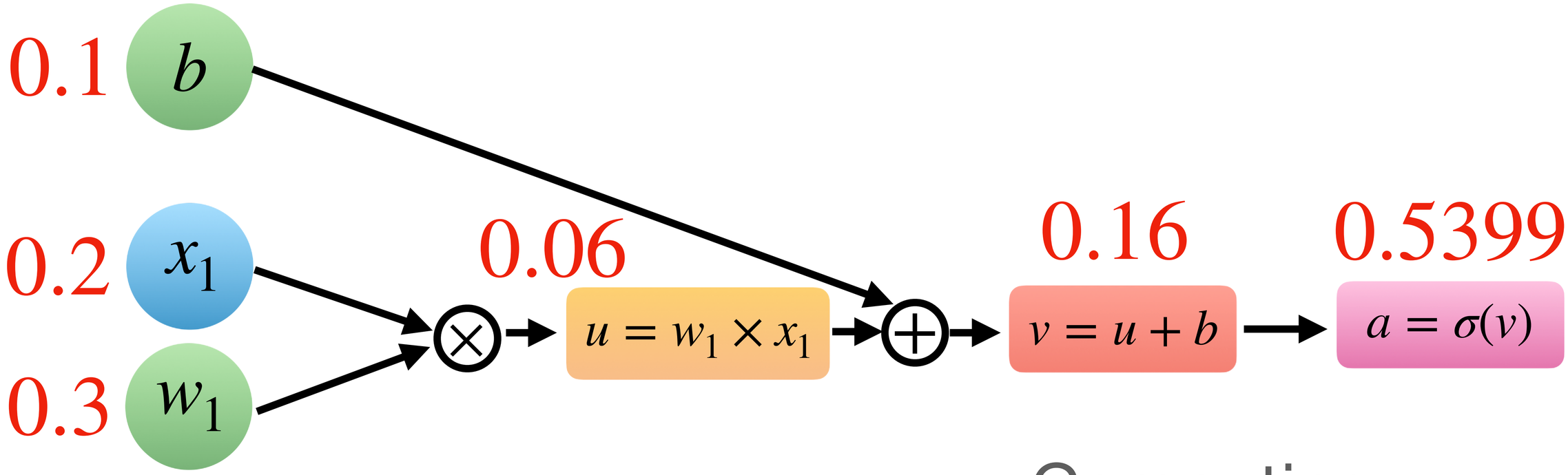

0.1
b
0.2
x1
0.3
w1
0.06
u = w1 × x1
0.16
v = u + b
0.5399
a = σ(v)



Forward pass in PyTorch

```
b = torch.tensor(0.1)
x1 = torch.tensor(0.2)
w1 = torch.tensor(0.3)

u = w1*x1
v = u + b
a = torch.sigmoid(v)
a
```

```
tensor(0.5399)
```




```
b = torch.tensor(0.1)
x1 = torch.tensor(0.2)
w1 = torch.tensor(0.3)

u = w1*x1
v = u + b
a = torch.sigmoid(v)
a
```

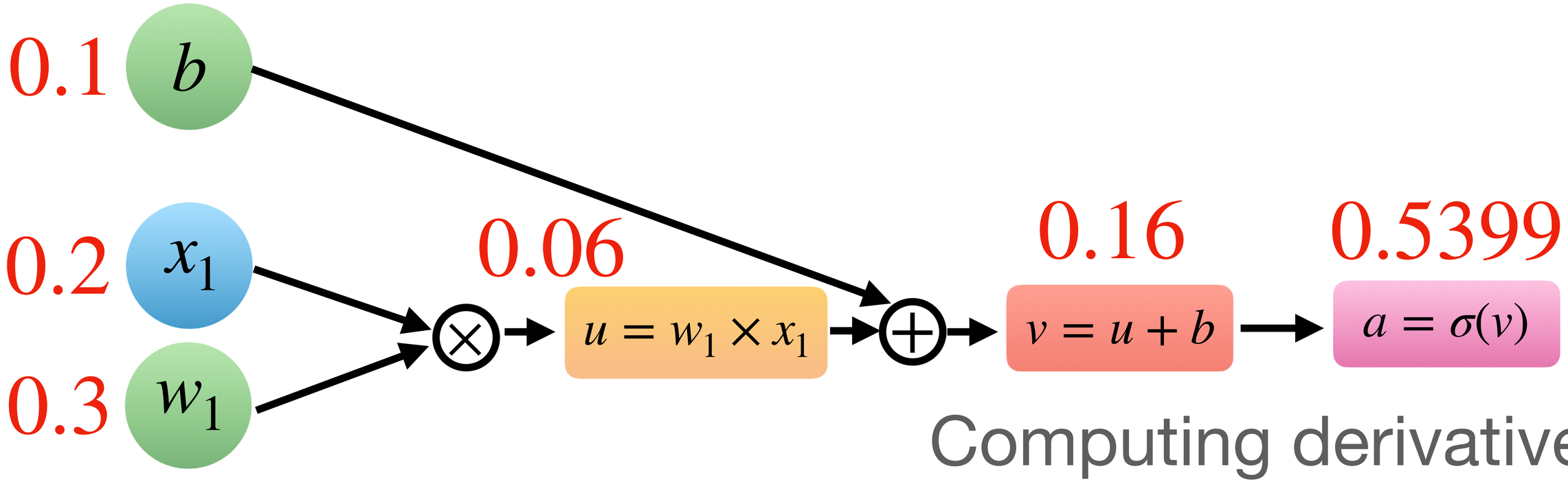
```
tensor(0.5399)
```

Computing derivatives manually

$$\frac{da}{dw_1} =$$

```
a * (1-a) * x1
```

```
tensor(0.0497)
```



Computing derivatives **automatically!**

```
b = torch.tensor(0.1)
x1 = torch.tensor(0.2)
w1 = torch.tensor(0.3, requires_grad=True)

u = w1*x1
v = u + b
a = torch.sigmoid(v)
a
```

```
tensor(0.5399, grad_fn=<SigmoidBackward0>)
```

$$\frac{da}{dw_1} =$$

```
from torch.autograd import grad

grad(a, w1)
```

```
(tensor(0.0497),)
```

3

Deep learning library

What is PyTorch?

Neural network training in 3 steps!

Step 1: Defining the dataset

Step 2: Defining the model

Step 3: Defining the training loop

Focus for today:

Step 1: Defining the model

Step 2: Defining the training loop

Step 1: Defining the model

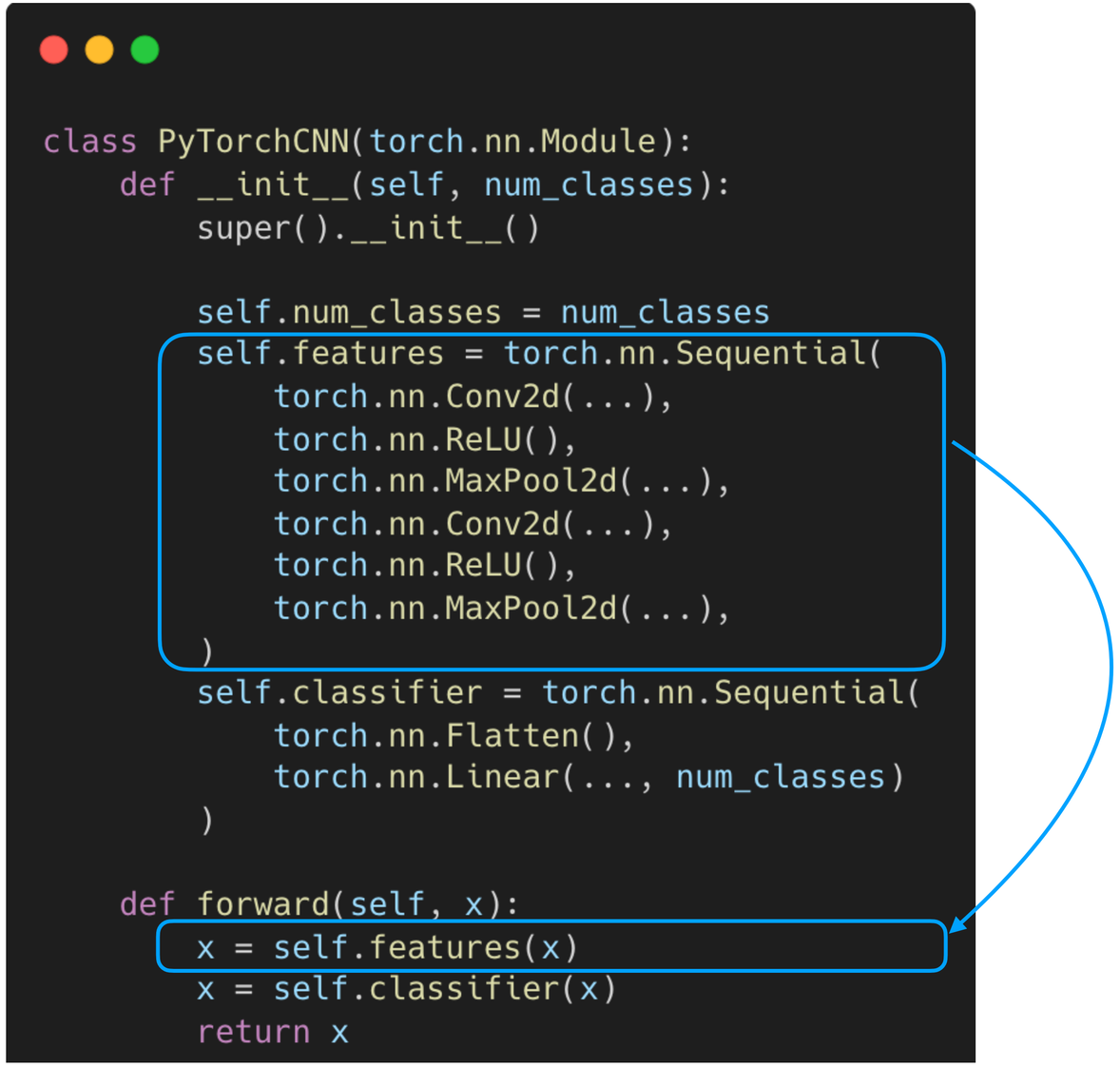
Define layers

```
class PyTorchCNN(torch.nn.Module):
    def __init__(self, num_classes):
        super().__init__()

        self.num_classes = num_classes
        self.features = torch.nn.Sequential(
            torch.nn.Conv2d(...),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(...),
            torch.nn.Conv2d(...),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(...),
        )
        self.classifier = torch.nn.Sequential(
            torch.nn.Flatten(),
            torch.nn.Linear(..., num_classes)
        )
```

Define forward method

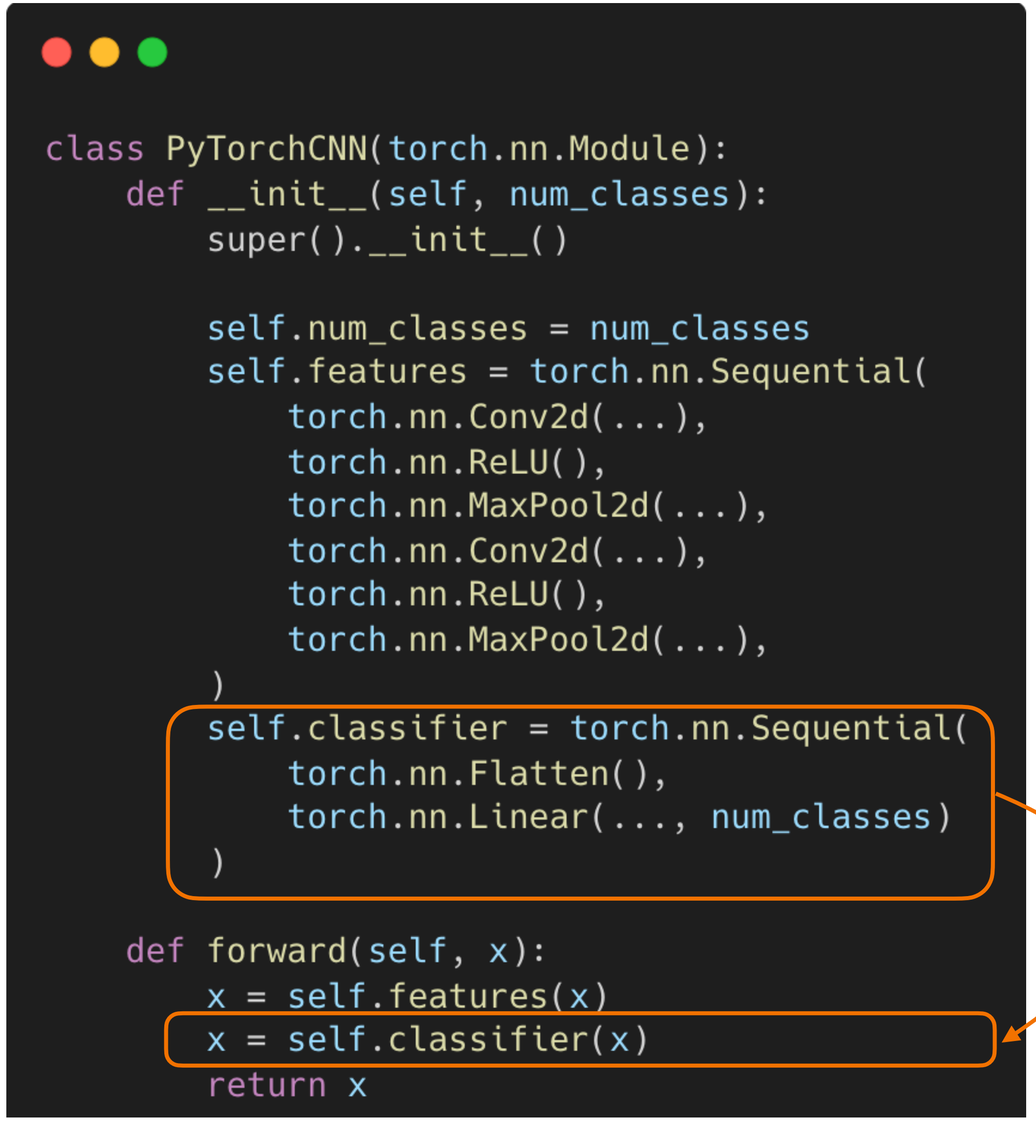
```
class PyTorchCNN(torch.nn.Module):
    def __init__(self, num_classes):
        super().__init__()

        self.num_classes = num_classes
        self.features = torch.nn.Sequential(
            torch.nn.Conv2d(...),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(...),
            torch.nn.Conv2d(...),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(...),
        )
        self.classifier = torch.nn.Sequential(
            torch.nn.Flatten(),
            torch.nn.Linear(..., num_classes)
        )

    def forward(self, x):
        x = self.features(x)
        x = self.classifier(x)
        return x
```

Definition order equals execution order



```
class PyTorchCNN(torch.nn.Module):
    def __init__(self, num_classes):
        super().__init__()

        self.num_classes = num_classes
        self.features = torch.nn.Sequential(
            torch.nn.Conv2d(...),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(...),
            torch.nn.Conv2d(...),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(...),
        )
        self.classifier = torch.nn.Sequential(
            torch.nn.Flatten(),
            torch.nn.Linear(..., num_classes)
        )

    def forward(self, x):
        x = self.features(x)
        x = self.classifier(x)
        return x
```

Definition order equals execution order



```
class PyTorchCNN(torch.nn.Module):
    def __init__(self, num_classes):
        super().__init__()

        self.num_classes = num_classes
        self.features = torch.nn.Sequential(
            torch.nn.Conv2d(...),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(...),
            torch.nn.Conv2d(...),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(...),
        )
        self.classifier = torch.nn.Sequential(
            torch.nn.Flatten(),
            torch.nn.Linear(..., num_classes)
        )

    def forward(self, x):
        x = self.features(x)
        x = self.classifier(x)
        return x
```

You are free to customize the forward method

Same network as before but defines computation in forward

```
class PyTorchCNN(torch.nn.Module):
    def __init__(self, num_classes):
        super().__init__()

        self.conv_1 = torch.nn.Conv2d(...)
        self.conv_2 = torch.nn.Conv2d(...)
        self.linear_1 = torch.nn.Linear(..., num_classes)

    def forward(self, x):
        out = self.conv_1(x)
        out = torch.nn.functional.relu(out)
        out = torch.nn.functional.max_pool2d(out)

        out = self.conv_2(out)
        out = torch.nn.functional.relu(out)
        out = torch.nn.functional.max_pool2d(out)

        out = torch.flatten(out, dim=1)
        logits = self.linear_1(out)
        return logits
```

Step 2: Defining the training loop

Initializing the model and optimizer

```
model = PyTorchCNN(num_classes=num_classes)
model = model.to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

Iterating over the training examples

```
model = PyTorchCNN(num_classes=num_classes)
model = model.to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)


for epoch in range(num_epochs):
    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):

        features, targets = features.to(device), targets.to(device)
```

Computing the predictions

```
model = PyTorchCNN(num_classes=num_classes)
model = model.to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)


for epoch in range(num_epochs):
    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):

        features, targets = features.to(device), targets.to(device)

        ### Forward pass
        logits = model(features)
        loss = F.cross_entropy(logits, targets)
```

Computing the predictions

```
model = PyTorchCNN(num_classes=num_classes)
model = model.to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)


for epoch in range(num_epochs):
    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):

        features, targets = features.to(device), targets.to(device)

        ### Forward pass
        logits = model(features)
        loss = F.cross_entropy(logits, targets)
```

the `cross_entropy` loss takes care of the LogSoftmax internally

Computing the predictions

```
model = PyTorchCNN(num_classes=num_classes)
model = model.to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)


for epoch in range(num_epochs):
    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):

        features, targets = features.to(device), targets.to(device)

        ### Forward pass
        logits = model(features)
        loss = F.cross_entropy(logits, targets)
```

the `cross_entropy` loss also takes care of one-hot encoding internally

Computing the backward pass

```
model = PyTorchCNN(num_classes=num_classes)
model = model.to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)


for epoch in range(num_epochs):
    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):

        features, targets = features.to(device), targets.to(device)

        ### Forward pass
        logits = model(features)
        loss = F.cross_entropy(logits, targets)

        ### Backward pass (backpropagation)
        optimizer.zero_grad()
        loss.backward()
```

Computing the backward pass

```
model = PyTorchCNN(num_classes=num_classes)
model = model.to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)


for epoch in range(num_epochs):
    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):

        features, targets = features.to(device), targets.to(device)

        ### Forward pass
        logits = model(features)
        loss = F.cross_entropy(logits, targets)

        ### Backward pass (backpropagation)
        optimizer.zero_grad()
        loss.backward()
```

If you omit this, gradients will be accumulated, which we usually don't want

Updating the model weights

```
model = PyTorchCNN(num_classes=num_classes)
model = model.to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)


for epoch in range(num_epochs):
    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):

        features, targets = features.to(device), targets.to(device)

        ### Forward pass
        logits = model(features)
        loss = F.cross_entropy(logits, targets)

        ### Backward pass (backpropagation)
        optimizer.zero_grad()
        loss.backward()

        ### Update model parameters
        optimizer.step()
```

Tracking the perfor-mance

```
model = PyTorchCNN(num_classes=num_classes)
model = model.to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)


for epoch in range(num_epochs):
    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):

        features, targets = features.to(device), targets.to(device)

        ### Forward pass
        logits = model(features)
        loss = F.cross_entropy(logits, targets)

        ### Backward pass (backpropagation)
        optimizer.zero_grad()
        loss.backward()

        ### Update model parameters
        optimizer.step()

    ### Optional evaluation steps
    model.eval()
    with torch.no_grad():
        valid_acc = compute_accuracy(model, valid_loader, device)
        print(f"Validation accuracy: {valid_acc * 100:.2f}%")
```

.train()
& **.eval()**
modes
matter
for things like
BatchNorm,
Dropout etc.

```
model = PyTorchCNN(num_classes=num_classes)
model = model.to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)


for epoch in range(num_epochs):
    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):

        features, targets = features.to(device), targets.to(device)

        ### Forward pass
        logits = model(features)
        loss = F.cross_entropy(logits, targets)

        ### Backward pass (backpropagation)
        optimizer.zero_grad()
        loss.backward()

        ### Update model parameters
        optimizer.step()

    ### Optional evaluation steps
    model.eval()
    with torch.no_grad():
        valid_acc = compute_accuracy(model, valid_loader, device)
        print(f"Validation accuracy: {valid_acc * 100:.2f}%")
```

no_grad() prevents unnecessary graph construction

```
model = PyTorchCNN(num_classes=num_classes)
model = model.to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)


for epoch in range(num_epochs):
    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):

        features, targets = features.to(device), targets.to(device)

        ### Forward pass
        logits = model(features)
        loss = F.cross_entropy(logits, targets)

        ### Backward pass (backpropagation)
        optimizer.zero_grad()
        loss.backward()

        ### Update model parameters
        optimizer.step()

    ### Optional evaluation steps
    model.eval()
    with torch.no_grad():
        valid_acc = compute_accuracy(model, valid_loader, device)
        print(f"Validation accuracy: {valid_acc * 100:.2f}%")
```

Why do I like PyTorch?

It’s Pythonic and flexible!

A custom layer

```
import torch


class CoralLayer(torch.nn.Module):
    def __init__(self, size_in, num_classes, preinit_bias=True):
        super().__init__()
        self.size_in, self.size_out = size_in, 1

        self.coral_weights = torch.nn.Linear(self.size_in, 1, bias=False)
        if preinit_bias:
            self.coral_bias = torch.nn.Parameter(
                torch.arange(num_classes - 1, 0, -1).float() / (num_classes-1))
        else:
            self.coral_bias = torch.nn.Parameter(
                torch.zeros(num_classes-1).float())

    def forward(self, x):
        return self.coral_weights(x) + self.coral_bias
```

https://github.com/Raschka-research-group/coral-pytorch

A Keras port of the custom layer

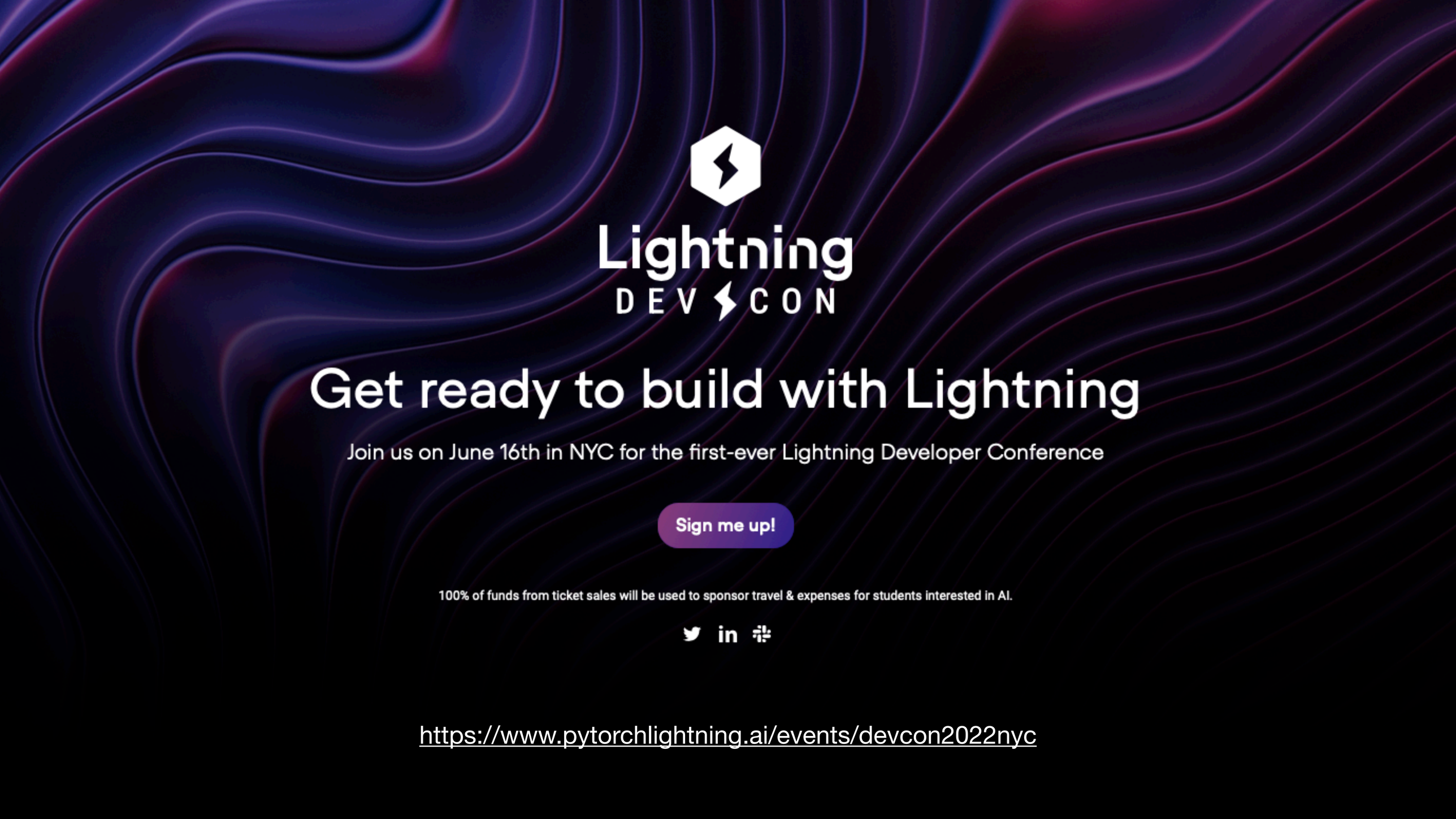
```python
from typing import Optional
import warnings
import tensorflow as tf
import tensorflow.keras.regularizers


@tf.keras.utils.register_keras_serializable(package="coral_ordinal")
class CoralOrdinal(tf.keras.layers.Layer):

    def __init__(
        self,
        num_classes: int,
        activation: Optional[str] = None,
        kernel_regularizer: Optional[tf.keras.regularizers.Regularizer] = None,
        bias_regularizer: Optional[tf.keras.regularizers.Regularizer] = None,
        **kwargs,
    ):

        if "input_shape" not in kwargs and "input_dim" in kwargs:
            kwargs["input_shape"] = (kwargs.pop("input_dim"),)
        super(CoralOrdinal, self).__init__(**kwargs)
        self.num_classes = num_classes
        self.activation = tf.keras.activations.get(activation)
        self.kernel_regularizer = tf.keras.regularizers.get(kernel_regularizer)
        self.bias_regularizer = tf.keras.regularizers.get(bias_regularizer)

    def build(self, input_shape):
        num_units = 1

        self.kernel = self.add_weight(
            shape=(input_shape[-1], num_units),
            name=self.name + "_latent",
            initializer="glorot_uniform",
            regularizer=self.kernel_regularizer,
            dtype=tf.float32,
            trainable=True,
        )

        self.bias = self.add_weight(
            shape=(self.num_classes - 1,),
            name=self.name + "_bias",
            regularizer=self.bias_regularizer,
            initializer="zeros",
            dtype=tf.float32,
            trainable=True,
        )

    def call(self, inputs):
        kernelized_inputs = tf.matmul(inputs, self.kernel)

        logits = kernelized_inputs + self.bias

        if self.activation is None:
            outputs = logits
        else:
            outputs = self.activation(logits)

        return outputs

    def get_config(self):
        config = super(CoralOrdinal, self).get_config()
        config.update(
            {
                "num_classes": self.num_classes,
                "kernel_regularizer": self.kernel_regularizer,
                "bias_regularizer": self.bias_regularizer,
            }
        )
        return config
```

https://github.com/ck37/coral-ordinal/blob/master/coral_ordinal/layer.py

Live Demo

https://sebastianraschka.com/books/

https://github.com/rasbt/machine-learning-book



https://www.pytorchlightning.ai/events/devcon2022nyc

Live Demo

https://github.com/PyTorchLightning/dataumbrella22-intro-pytorch

Introduction to PyTorch

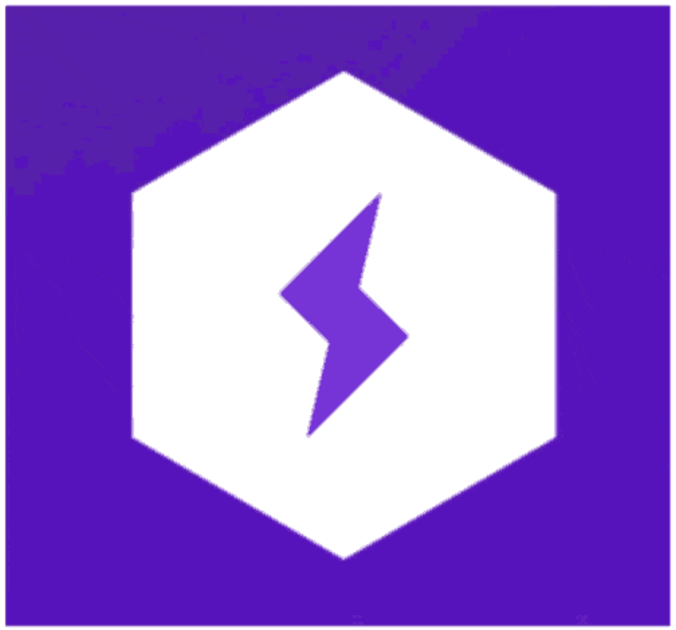
and Scaling PyTorch Code Using LightningLite

Sebastian Raschka & Adrian Wälchli

Data Umbrella

May 10th, 2022

What lies ahead of you

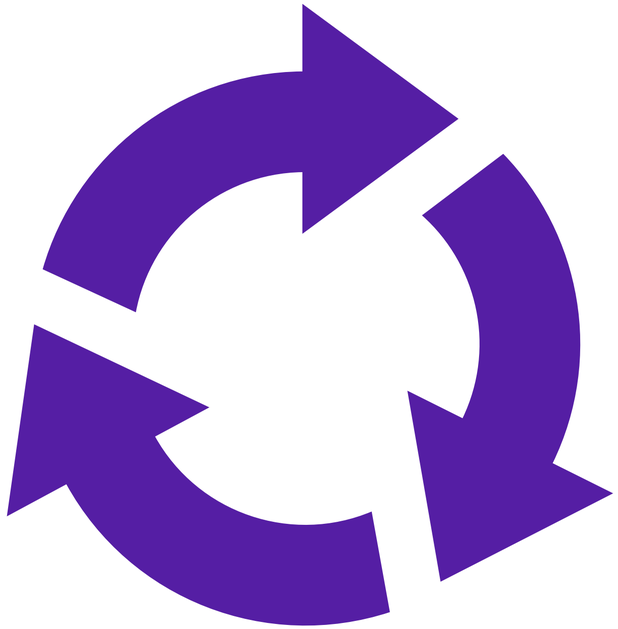


Lightning Lite

Powered by Lightning Accelerators

Idea

Test



Code / Debug

Run Experiments

Idea

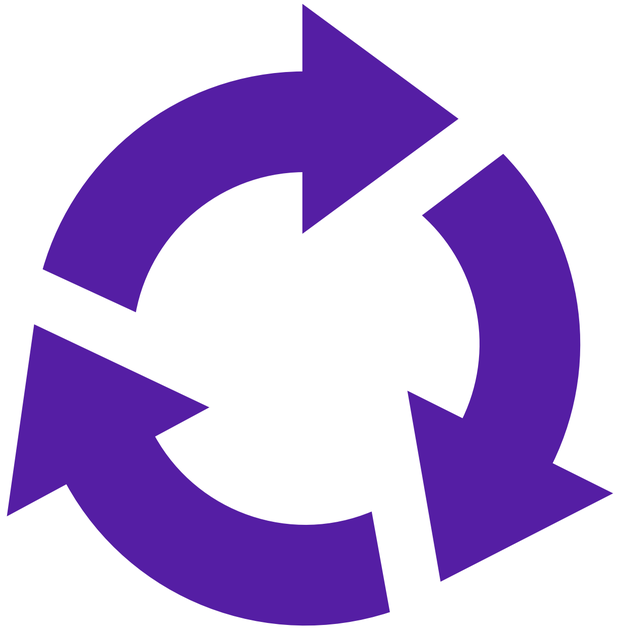
Lots of Boilerplate

Code / Debug

Run Experiments

Many hours!

Test



What's boilerplate code?

```
    if args.gpu is not None:
        images = images.cuda(args.gpu, non_blocking=True)
    if torch.cuda.is_available():
        target = target.cuda(args.gpu, non_blocking=True)
```

https://github.com/pytorch/examples/blob/main/imagenet/main.py

What's boilerplate code?

```
    if args.distributed:
        if args.dist_url == "env://" and args.rank == -1:
            args.rank = int(os.environ["RANK"])
        if args.multiprocessing_distributed:
            # For multiprocessing distributed training, rank needs to be the
            # global rank among all the processes
            args.rank = args.rank * ngpus_per_node + gpu
        dist.init_process_group(backend=args.dist_backend,
                                init_method=args.dist_url,
                                world_size=args.world_size,
                                rank=args.rank)
```

https://github.com/pytorch/examples/blob/main/imagenet/main.py

What's boilerplate code?

```
    if args.distributed:
        train_sampler = torch.utils.data.distributed.DistributedSampler(train_dataset)
    else:
        train_sampler = None

    train_loader = torch.utils.data.DataLoader(
        train_dataset, batch_size=args.batch_size, shuffle=(train_sampler is None),
        num_workers=args.workers, pin_memory=True, sampler=train_sampler
    )
```

https://github.com/pytorch/examples/blob/main/imagenet/main.py

What does LightningLite do?

It handles all this boilerplate for you!

You get

CPU, GPU, multi-GPU, TPU
Multi-node
Mixed precision

For FREE, without the boilerplate

Let's do it!

```
pip install pytorch-lightning
```

PyTorch Lightning

www.pytorchlightning.ai

No changes to the model required!

```python
import torch


class PyTorchCNN(torch.nn.Module):
    def __init__(self, num_classes):
        super().__init__()

        self.num_classes = num_classes
        self.features = torch.nn.Sequential(
            torch.nn.Conv2d(
                in_channels=3,
                out_channels=8,
                kernel_size=(3, 3),
                stride=(1, 1),
                padding=1,
            ),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2), padding=0),
            torch.nn.ReLU(),
            torch.nn.Conv2d(
                in_channels=8,
                out_channels=16,
                kernel_size=(3, 3),
                stride=(1, 1),
                padding=1,
            ),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2), padding=0),
        )

        self.classifier = torch.nn.Sequential(
            torch.nn.Flatten(),
            torch.nn.Linear(784, 128),
            torch.nn.ReLU(),
            torch.nn.Linear(128, num_classes),
        )

    def forward(self, x):
        x = self.features(x)
        x = self.classifier(x)
        return x
```

No changes to the data required!

```python
from torch.utils.data import DataLoader

train_loader = DataLoader(
    dataset=train_dset,
    batch_size=batch_size,
    drop_last=True,
    num_workers=4,
    shuffle=True,
)

valid_loader = DataLoader(
    dataset=valid_dset,
    batch_size=batch_size,
    drop_last=False,
    num_workers=4,
    shuffle=False,
)

test_loader = DataLoader(
    dataset=test_dset,
    batch_size=batch_size,
    drop_last=False,
    num_workers=4,
    shuffle=False,
)
```

The LightningLite Skeleton

```
from pytorch_lightning.lite import LightningLite


class Lite(LightningLite):
    def run(self):

        # Here goes the training code


Lite().run()
```

Initializing the model and optimizer

Sets up model and optimizer for distributed training!

```
class Lite(LightningLite):
    def run(self):

        model = PyTorchCNN(num_classes=num_classes)
        model = model.to(device)
        optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

        model, optimizer = self.setup(model, optimizer)

        # ...
```

Setting up the data loaders

Automatically moves the data to the right device!

```
class Lite(LightningLite):
    def run(self):

        model = PyTorchCNN(num_classes=num_classes)
        optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

        model, optimizer = self.setup(model, optimizer)

        train_dataloader = self.setup_dataloaders(train_dataloader)
        val_dataloader = self.setup_dataloaders(val_dataloader)
        teset_dataloader = self.setup_dataloaders(test_dataloader)

        # ...
```

Iterating over the training examples

The features and targets are already on the device!

```
class Lite(LightningLite):
    def run(self):

        # ...

        for epoch in range(num_epochs):
            model = model.train()
            for batch_idx, (features, targets) in enumerate(train_loader):

                features, targets = features.to(device), targets.to(device)

                # ...
```

Updating the model weights

You only need to replace
loss.backward()
with
self.backward(loss)

```
class Lite(LightningLite):
    def run(self):

        # ...

        for epoch in range(num_epochs):
            model = model.train()
            for batch_idx, (features, targets) in enumerate(train_loader):

                ### Forward pass
                logits = model(features)
                loss = F.cross_entropy(logits, targets)

                ### Backward pass (backpropagation)
                optimizer.zero_grad()
                loss.backward()
                self.backward(loss)

                ### Update model parameters
                optimizer.step()

                # ...
```

We're done. Why did we do this again?

Accelerate your PyTorch code!

```
# Everything on CPU
Lite().run()

# One GPU
Lite(accelerator="gpu", devices=1).run()

# Multiple GPUs
Lite(accelerator="gpu", devices=4).run()

# Specific GPU IDs
Lite(accelerator="gpu", devices=[2, 3]).run()

# TPU
Lite(accelerator="tpu", devices=8).run()

# Select available hardware automatically!
Lite(accelerator="auto", devices="auto").run()
```

Mixed Precision saves you memory

```
# Default precision setting is 32-bit
Lite(accelerator="gpu", devices=1, precision=32).run()

# Save memory with mixed 16-bit precision
Lite(accelerator="gpu", devices=1, precision=16).run()

# Double precision is also supported
Lite(accelerator="gpu", devices=1, precision=64).run()
```

Try different strategies for best performance

```
# Best for multi-GPU in Jupyter notebooks
Lite(accelerator="gpu", devices=2, strategy="dp").run()

# Best for single and multi-node training
Lite(accelerator="gpu", devices=4, strategy="ddp").run()

# Sharded training saves memory for very large models!
Lite(accelerator="gpu", devices=8, strategy="ddp_sharded").run()

# For even bigger models
Lite(accelerator="gpu", devices=8, strategy="deepspeed").run()
```

When you're ready, level up in Lightning.

IPU Accelerator

Fault-tolerance

Progress Bar

Mixed Precision

Multi-GPU

Reproducibility

Logging

Checkpointing

Hyperparameter Tuner

Profiling

Cloud Computing

Metrics

Multi-Node

Loops

CLI

Gradient Accumulation

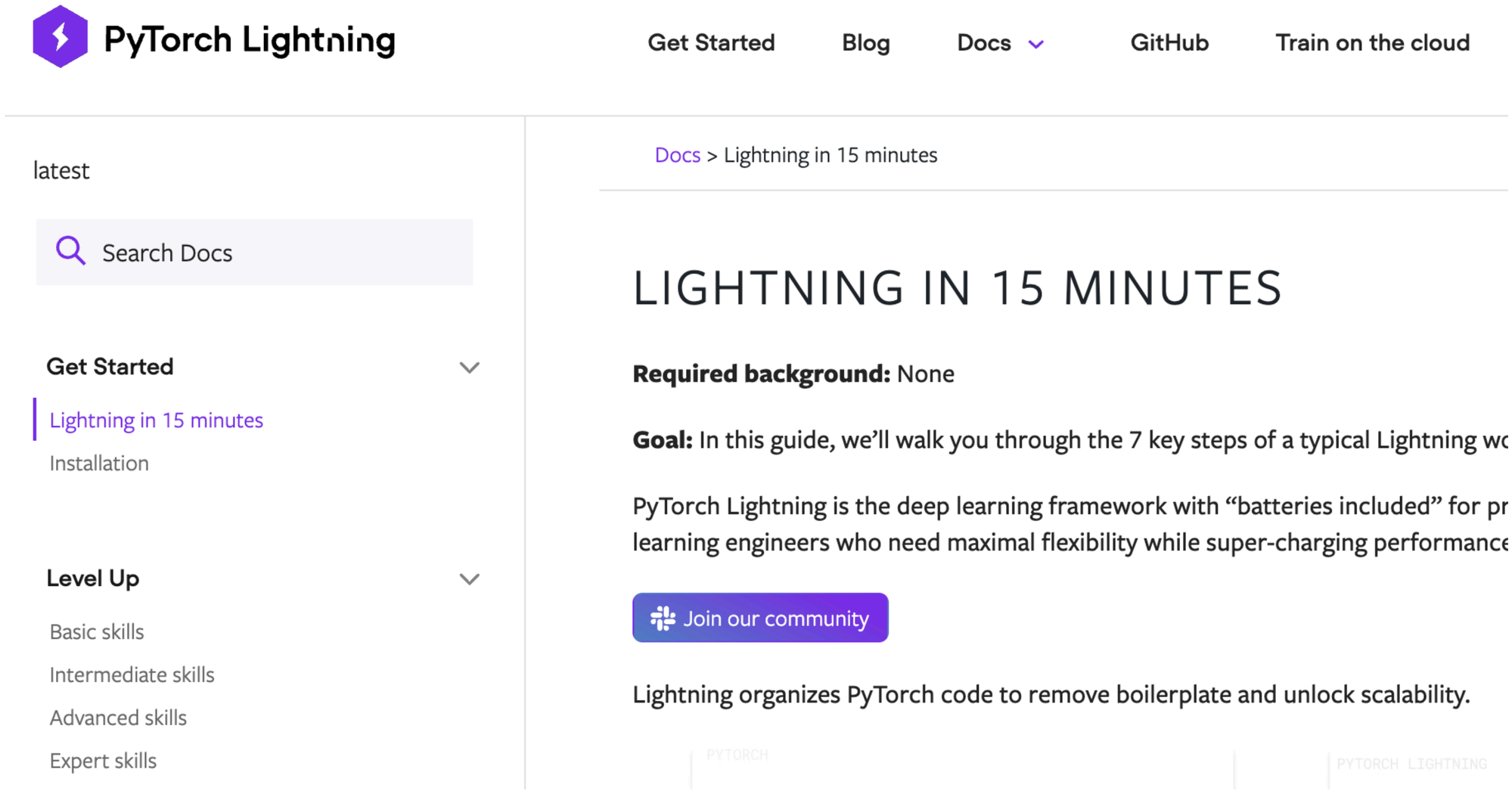
Early Stopping

Quantization

TPU Accelerator

HPU Accelerator

Visit docs.pytorchlightning.ai



Why is Lightning not in PyTorch?

The End

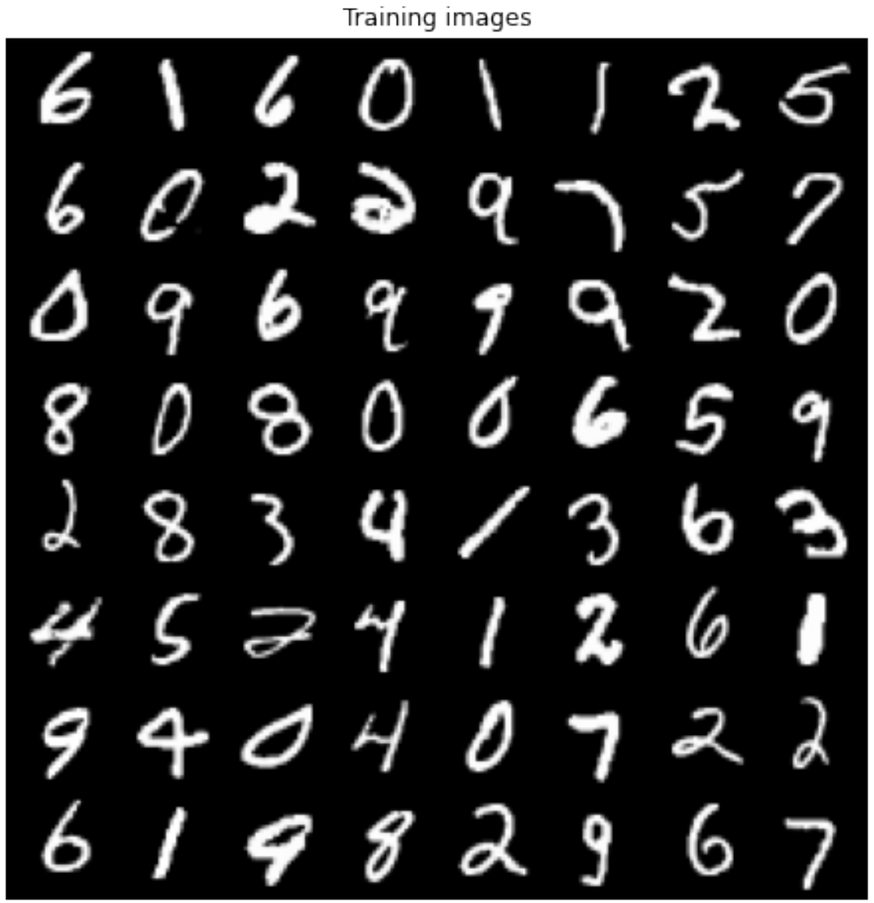
Supplementary

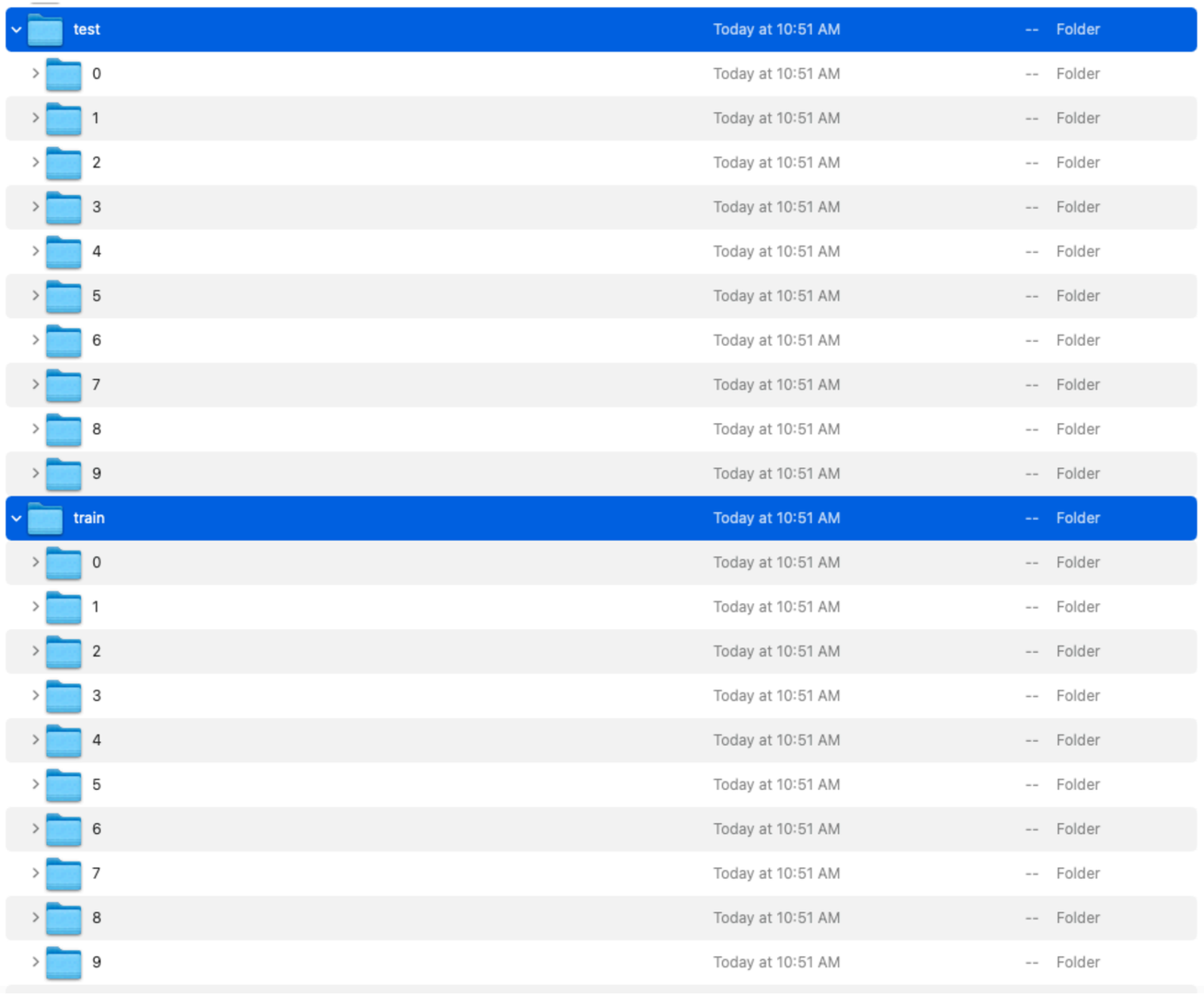
Step 3: Defining the dataset

MNIST: The “Hello World” of deep learning

Dataset source: http://yann.lecun.com/exdb/mnist/

MNIST:
The “Hello World” of deep learning





Building the dataset from folder structures

```
from torch.utils.data.dataset import random_split
from torchvision.datasets import ImageFolder

train_dset = ImageFolder(root="mnist-pngs/train", transform=data_transforms["train"])

train_dset, valid_dset = random_split(train_dset, lengths=[55000, 5000])

test_dset = ImageFolder(root="mnist-pngs/test", transform=data_transforms["test"])
```

Defining optional transformations

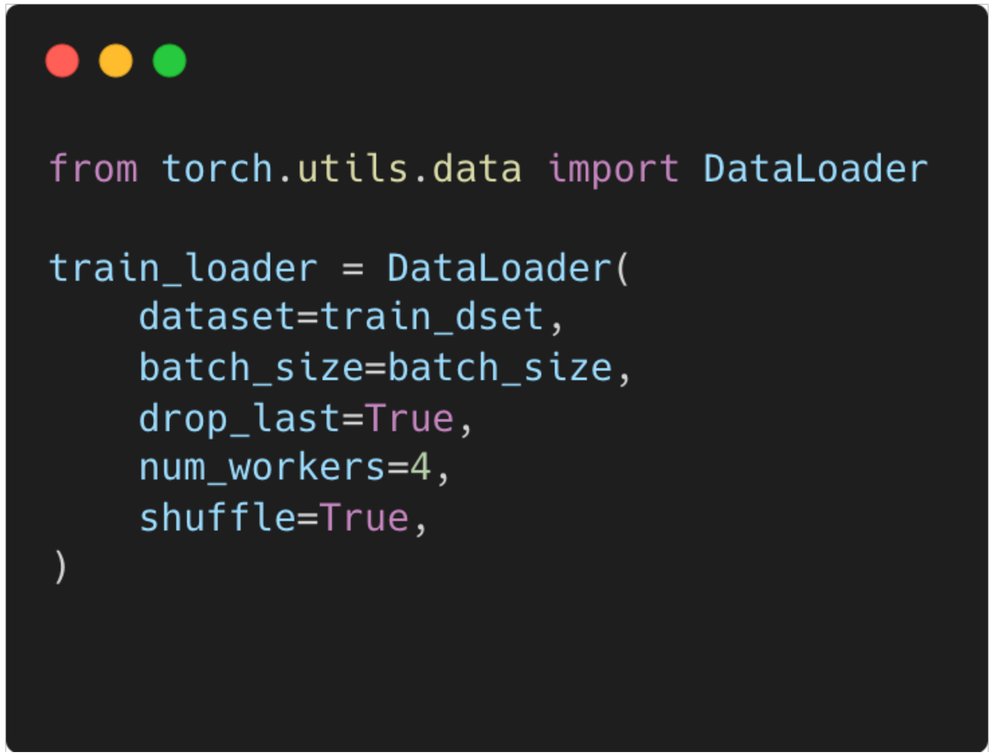
```python
from torch.utils.data.dataset import random_split
from torchvision.datasets import ImageFolder

train_dset = ImageFolder(root="mnist-pngs/train", transform=data_transforms["train"])

train_dset, valid_dset = random_split(train_dset, lengths=[55000, 5000])

test_dset = ImageFolder(root="mnist-pngs/test", transform=data_transforms["test"])
```

```python
from torchvision import transforms

data_transforms = {
    "train": transforms.Compose(
        [
            transforms.Resize(32),
            transforms.RandomCrop((28, 28)),
            transforms.ToTensor(),
            # normalize images to [-1, 1] range
            transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
        ]
    ),
```

Defining optional transformations

```python
from torch.utils.data.dataset import random_split
from torchvision.datasets import ImageFolder

train_dset = ImageFolder(root="mnist-pngs/train", transform=data_transforms["train"])

train_dset, valid_dset = random_split(train_dset, lengths=[55000, 5000])

test_dset = ImageFolder(root="mnist-pngs/test", transform=data_transforms["test"])
```

```python
data_transforms = {
    # ...
    "test": transforms.Compose(
        [
            transforms.Resize((32, 32)),
            transforms.CenterCrop((28, 28)),
            transforms.ToTensor(),
            # normalize images to [-1, 1] range
            transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
        ]
    ),
}
```



```python
from torch.utils.data import DataLoader

train_loader = DataLoader(
    dataset=train_dset,
    batch_size=batch_size,
    drop_last=True,
    num_workers=4,
    shuffle=True,
)
```

```python
from torch.utils.data import DataLoader

train_loader = DataLoader(
    dataset=train_dset,
    batch_size=batch_size,
    drop_last=True,
    num_workers=4,
    shuffle=True,
)
```

```python
valid_loader = DataLoader(
    dataset=valid_dset,
    batch_size=batch_size,
    drop_last=False,
    num_workers=4,
    shuffle=False,
)
```

```python
from torch.utils.data import DataLoader

train_loader = DataLoader(
    dataset=train_dset,
    batch_size=batch_size,
    drop_last=True,
    num_workers=4,
    shuffle=True,
)
```

```python
valid_loader = DataLoader(
    dataset=valid_dset,
    batch_size=batch_size,
    drop_last=False,
    num_workers=4,
    shuffle=False,
)
```

```python
test_loader = DataLoader(
    dataset=test_dset,
    batch_size=batch_size,
    drop_last=False,
    num_workers=4,
    shuffle=False,
)
```