CPEN 400D: Deep Learning

Lecture 3 (I): PyTorch and Autograd

Renjie Liao

University of British Columbia

Winter, Term 2, 2022

Outline

- Basics in PyTorch
- Computational Graphs, Autograd, and Gradient Checking
- Creating Models
- Loading Data and Training Models

Outline

- **Basics in PyTorch**
- Computational Graphs, Autograd, and Gradient Checking
- Creating Models
- Loading Data and Training Models

PyTorch

PyTorch [1] is a deep learning framework (free and open-sourced under the modified BSD license) based on the Torch library, originally developed by Meta AI and now part of the Linux Foundation umbrella.

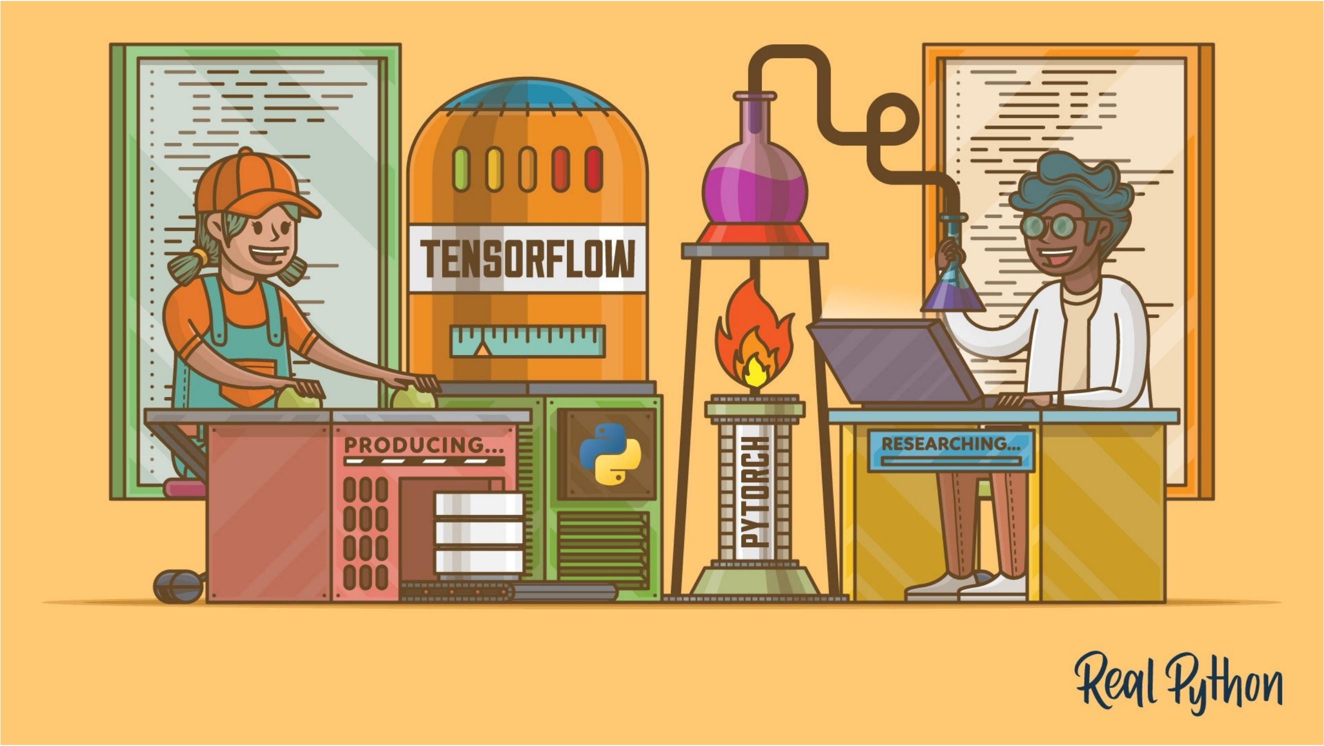


Image Credit: [2]

PyTorch

PyTorch [1] is a deep learning framework (free and open-sourced under the modified BSD license) based on the Torch library, originally developed by Meta AI and now part of the Linux Foundation umbrella.

Many pieces of deep learning software are built on top of PyTorch, including Tesla Autopilot, Uber's Pyro, Hugging Face's Transformers, PyTorch Lightning, and Catalyst.

Other prominent deep learning frameworks include JAX, Tensorflow, etc. You can find more in [3].

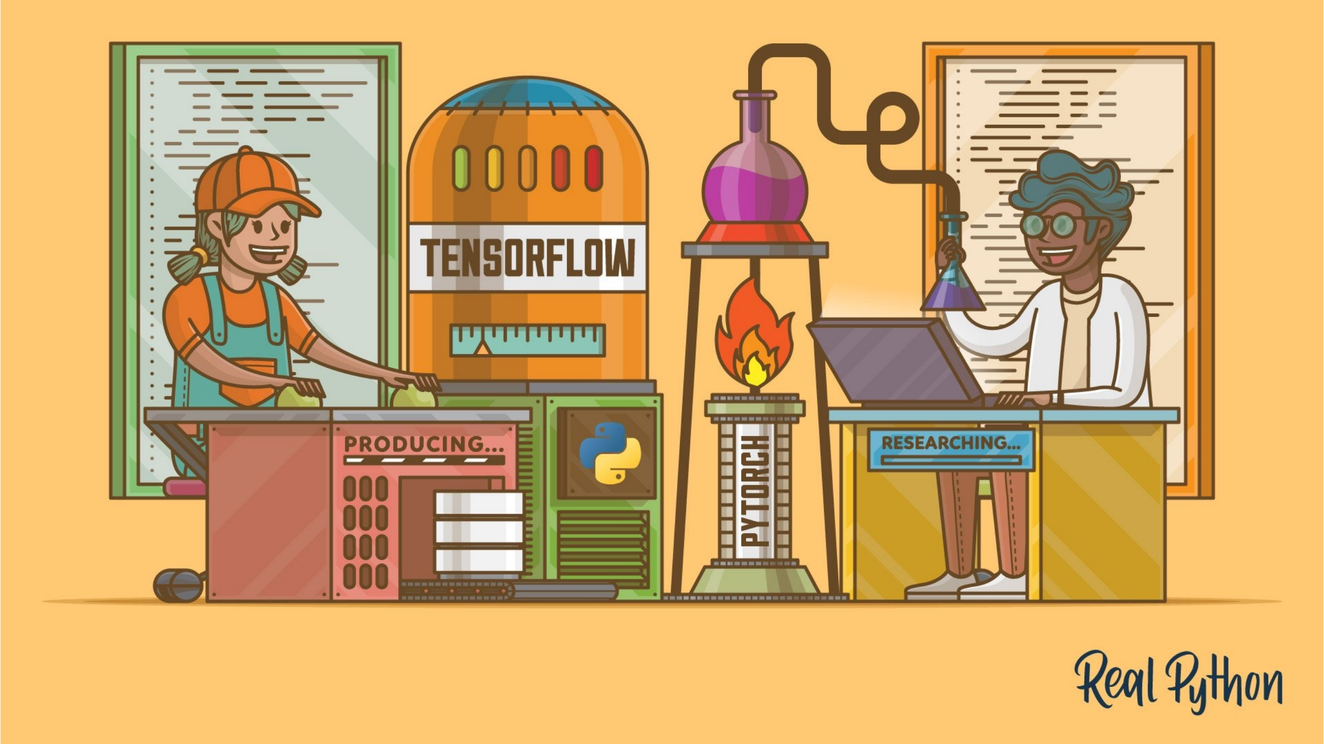
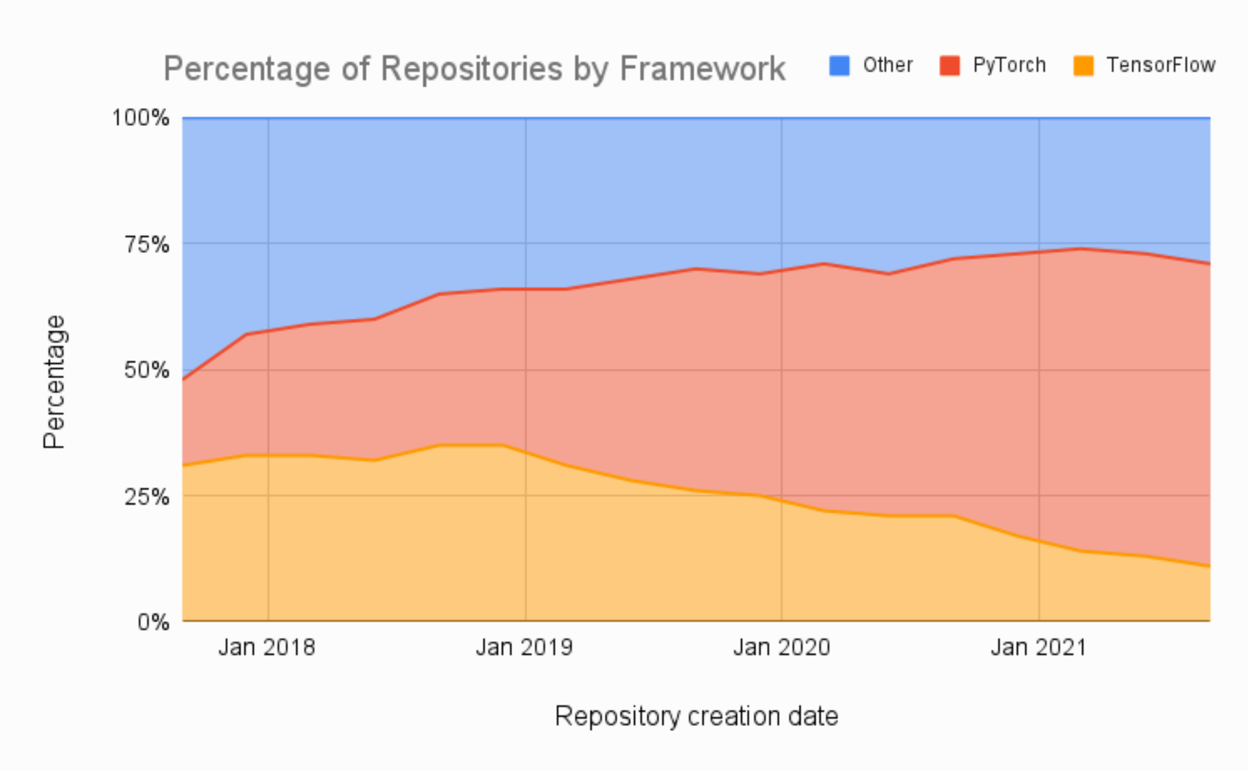


Image Credit: [2]

Popularity of PyTorch

PyTorch is popular (especially in “research”) and has a great ecosystem.



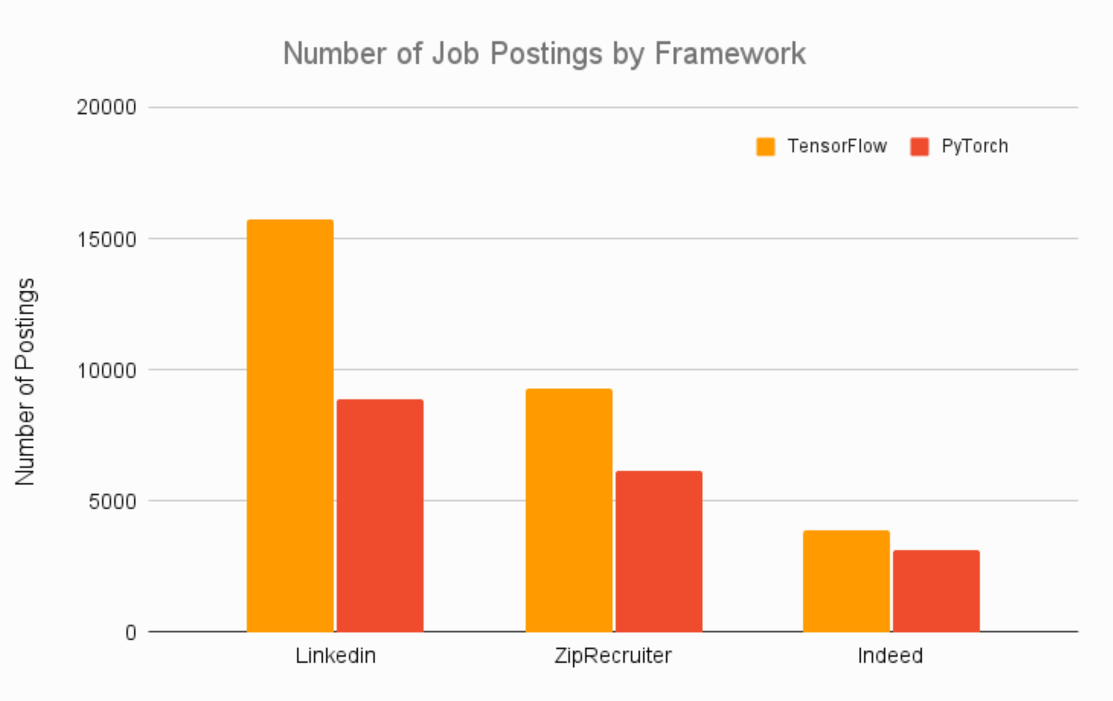


Image Credit: [4]

Design Principles in PyTorch

An excerpt of [1]: **Be Pythonic** Data scientists are familiar with the Python language, its programming model, and its tools. PyTorch should be a first-class member of that ecosystem. It follows the commonly established design goals of keeping interfaces simple and consistent, ideally with one idiomatic way of doing things. It also integrates naturally with standard plotting, debugging, and data processing tools.

Design Principles in PyTorch

An excerpt of [1]:

Be Pythonic Data scientists are familiar with the Python language, its programming model, and its tools. PyTorch should be a first-class member of that ecosystem. It follows the commonly established design goals of keeping interfaces simple and consistent, ideally with one idiomatic way of doing things. It also integrates naturally with standard plotting, debugging, and data processing tools.

Put researchers first PyTorch strives to make writing models, data loaders, and optimizers as easy and productive as possible. The complexity inherent to machine learning should be handled internally by the PyTorch library and hidden behind intuitive APIs free of side-effects and unexpected performance cliffs.

Design Principles in PyTorch

An excerpt of [1]:

Be Pythonic Data scientists are familiar with the Python language, its programming model, and its tools. PyTorch should be a first-class member of that ecosystem. It follows the commonly established design goals of keeping interfaces simple and consistent, ideally with one idiomatic way of doing things. It also integrates naturally with standard plotting, debugging, and data processing tools.

Put researchers first PyTorch strives to make writing models, data loaders, and optimizers as easy and productive as possible. The complexity inherent to machine learning should be handled internally by the PyTorch library and hidden behind intuitive APIs free of side-effects and unexpected performance cliffs.

Provide pragmatic performance To be useful, PyTorch needs to deliver compelling performance, although not at the expense of simplicity and ease of use. Trading 10% of speed for a significantly simpler to use model is acceptable; 100% is not. Therefore, its *implementation* accepts added complexity in order to deliver that performance. Additionally, providing tools that allow researchers to manually control the execution of their code will empower them to find their own performance improvements independent of those that the library provides automatically.

Design Principles in PyTorch

An excerpt of [1]:

Be Pythonic Data scientists are familiar with the Python language, its programming model, and its tools. PyTorch should be a first-class member of that ecosystem. It follows the commonly established design goals of keeping interfaces simple and consistent, ideally with one idiomatic way of doing things. It also integrates naturally with standard plotting, debugging, and data processing tools.

Put researchers first PyTorch strives to make writing models, data loaders, and optimizers as easy and productive as possible. The complexity inherent to machine learning should be handled internally by the PyTorch library and hidden behind intuitive APIs free of side-effects and unexpected performance cliffs.

Provide pragmatic performance To be useful, PyTorch needs to deliver compelling performance, although not at the expense of simplicity and ease of use. Trading 10% of speed for a significantly simpler to use model is acceptable; 100% is not. Therefore, its *implementation* accepts added complexity in order to deliver that performance. Additionally, providing tools that allow researchers to manually control the execution of their code will empower them to find their own performance improvements independent of those that the library provides automatically.

Worse is better Given a fixed amount of engineering resources, and all else being equal, the time saved by keeping the internal implementation of PyTorch simple can be used to implement additional features, adapt to new situations, and keep up with the fast pace of progress in the field of AI. Therefore it is better to have a simple but slightly incomplete solution than a comprehensive but complex and hard to maintain design.

PyTorch

- PyTorch wraps the backend (C/C++/CUDA) in a Python interface. You can write highly customized and efficient deep learning models directly in Python without worrying about the low-level implementation.
- PyTorch's eager execution evaluates tensor operations immediately and dynamically, thus supporting models on varying-size data well.
- Pytorch can be roughly viewed as Numpy with GPU supports. E.g. `torch.Tensor` is the basic object in PyTorch, similar to `numpy.array` in Numpy.

Python >>>

```python
>>> import torch
>>> import numpy as np

>>> x = np.array([[2., 4., 6.]])
>>> y = np.array([[1.], [3.], [5.]])

>>> m = torch.mul(torch.from_numpy(x), torch.from_numpy(y))

>>> m.numpy()
array([[ 2.,  4.,  6.],
       [ 6., 12., 18.],
       [10., 20., 30.]])
```


Image Credit: [2]

PyTorch

Let us look at another code snippet to get a sense of `torch.Tensor` and its operations:

```
import torch
dtype = torch.float
device = torch.device("cpu") # This executes all calculations on the CPU
# device = torch.device("cuda:0") # This executes all calculations on the GPU

# Creation of a tensor and filling of a tensor with random numbers
a = torch.randn(2, 3, device=device, dtype=dtype)
print(a) # Output of tensor A
# Output: tensor([[-1.1884,  0.8498, -1.7129],
#                 [-0.8816,  0.1944,  0.5847]])

# Creation of a tensor and filling of a tensor with random numbers
b = torch.randn(2, 3, device=device, dtype=dtype)
print(b) # Output of tensor B
# Output: tensor([[ 0.7178, -0.8453, -1.3403],
#                 [ 1.3262,  1.1512, -1.7070]])

print(a*b) # Output of a multiplication of the two tensors
# Output: tensor([[-0.8530, -0.7183,  2.58],
#                 [-1.1692,  0.2238, -0.9981]])

print(a.sum()) # Output of the sum of all elements in tensor A
# Output: tensor(-2.1540)

print(a[1,2]) # Output of the element in the third column of the second row (zero based)
# Output: tensor(0.5847)

print(a.max()) # Output of the maximum value in tensor A
# Output: tensor(-1.7129)
```


Image Credit: [5]

Outline

- Basics in PyTorch
- **Computational Graphs, Autograd, and Gradient Checking**
- Creating Models
- Loading Data and Training Models

Computational Graphs & Autograd

Let us look at the following example:

$$f(x, y) = \log(xy)$$

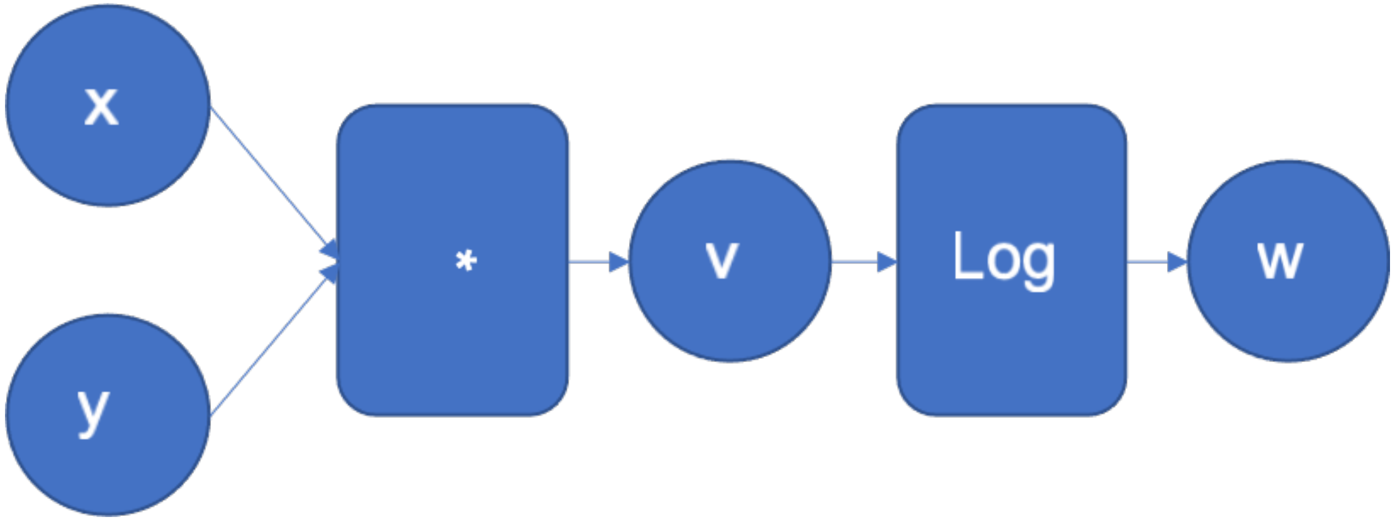


Image Credit: [6]

Computational Graphs & Autograd

Let us look at the following example:

$$f(x, y) = \log(xy)$$

Each operand (e.g., scalar, vector, matrix, or tensor) is a node and each operator is a node. The arrow represents the computational dependency. The computational graph is a directed acyclic graph (DAG).

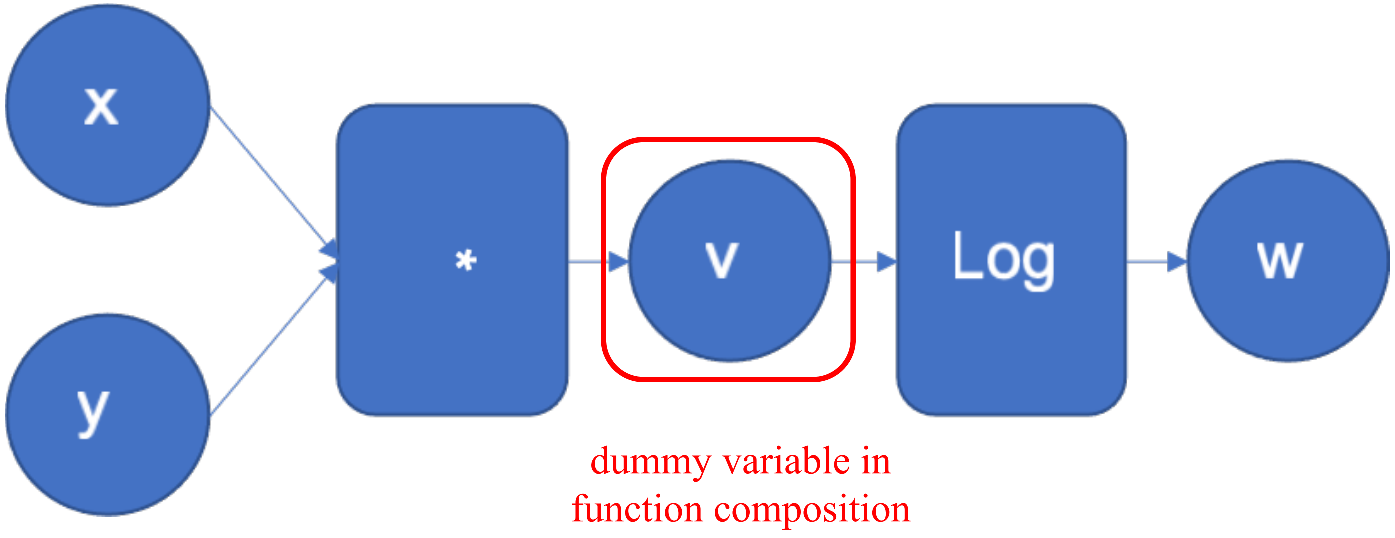


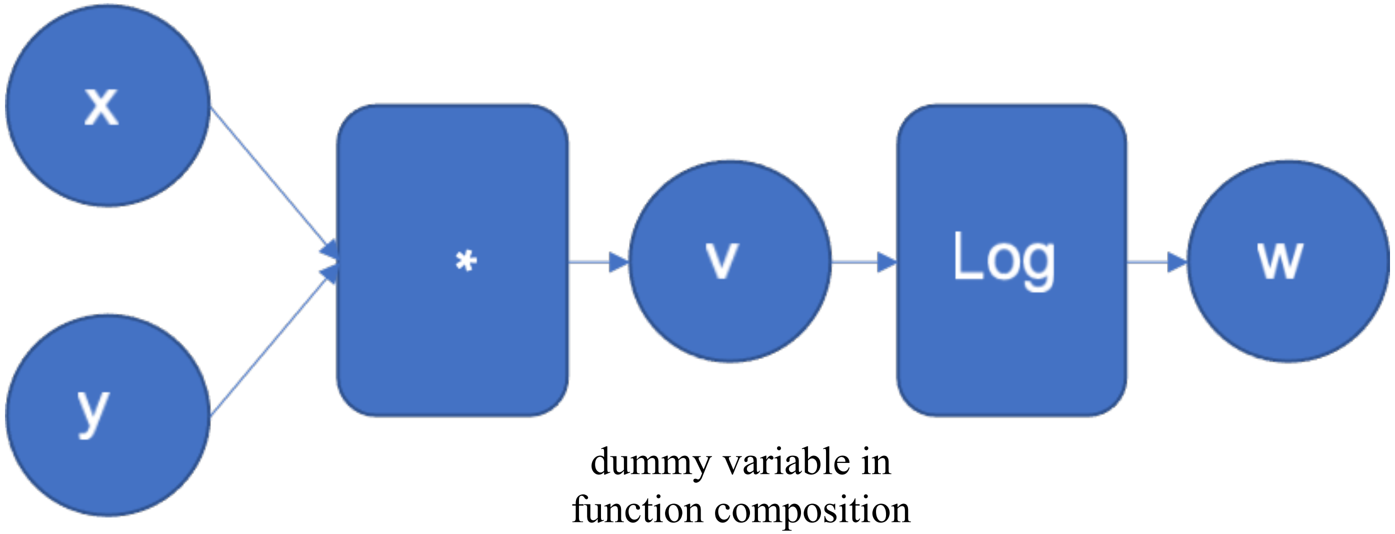
Image Credit: [6]

Computational Graphs & Autograd

Let us look at the following example:

$$f(x, y) = \log(xy)$$

Each operand (e.g., scalar, vector, matrix, or tensor) is a node and each operator is a node. The arrow represents the computational dependency. The computational graph is a directed acyclic graph (DAG).



dummy variable in
function composition

Sometimes one uses cycles to represent recurrent computations. But we can always unroll a recurrent computational graph as a DAG!

Image Credit: [6]

Computational Graphs & Autograd

Let us look at the following example:

$$f(x, y) = \log(xy)$$

Let us try it in PyTorch:

```
In [1]: import torch

In [2]: x = torch.tensor(0.5)

In [3]: y = torch.tensor(0.75)

In [4]: w = torch.log(x * y)

In [5]: w
Out[5]: tensor(-0.9808)
```

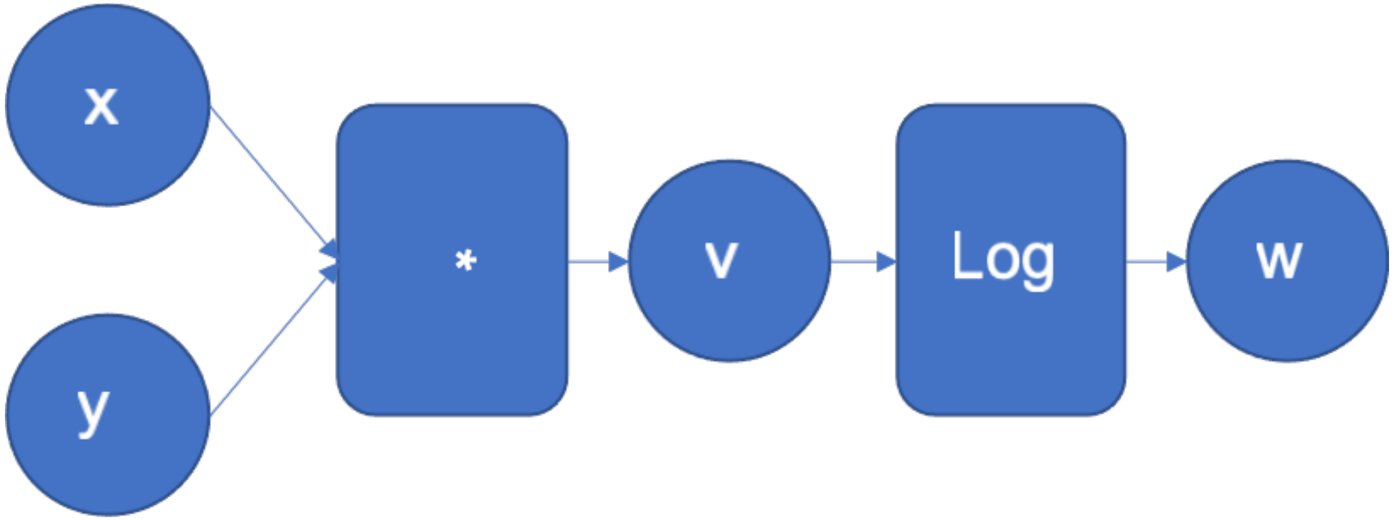


Image Credit: [6]

Computational Graphs & Autograd

If you need to compute gradients in the computational graph, you need to set the `requires_grad` attribute to be true for the tensor.

Every time Autograd (i.e., the automatic differentiation engine) executes an operation in the graph, the derivative of that operation is added to the graph to be executed later in the backward pass. Note that Autograd knows the derivatives of the basic functions.

Image Credit: [6]

Computational Graphs & Autograd

If you need to compute gradients in the computational graph, you need to set the `requires_grad` attribute to be true for the tensor.

$$f(x, y) = \log(xy)$$

Every time Autograd (i.e., the automatic differentiation engine) executes an operation in the graph, the derivative of that operation is added to the graph to be executed later in the backward pass. Note that Autograd knows the derivatives of the basic functions.

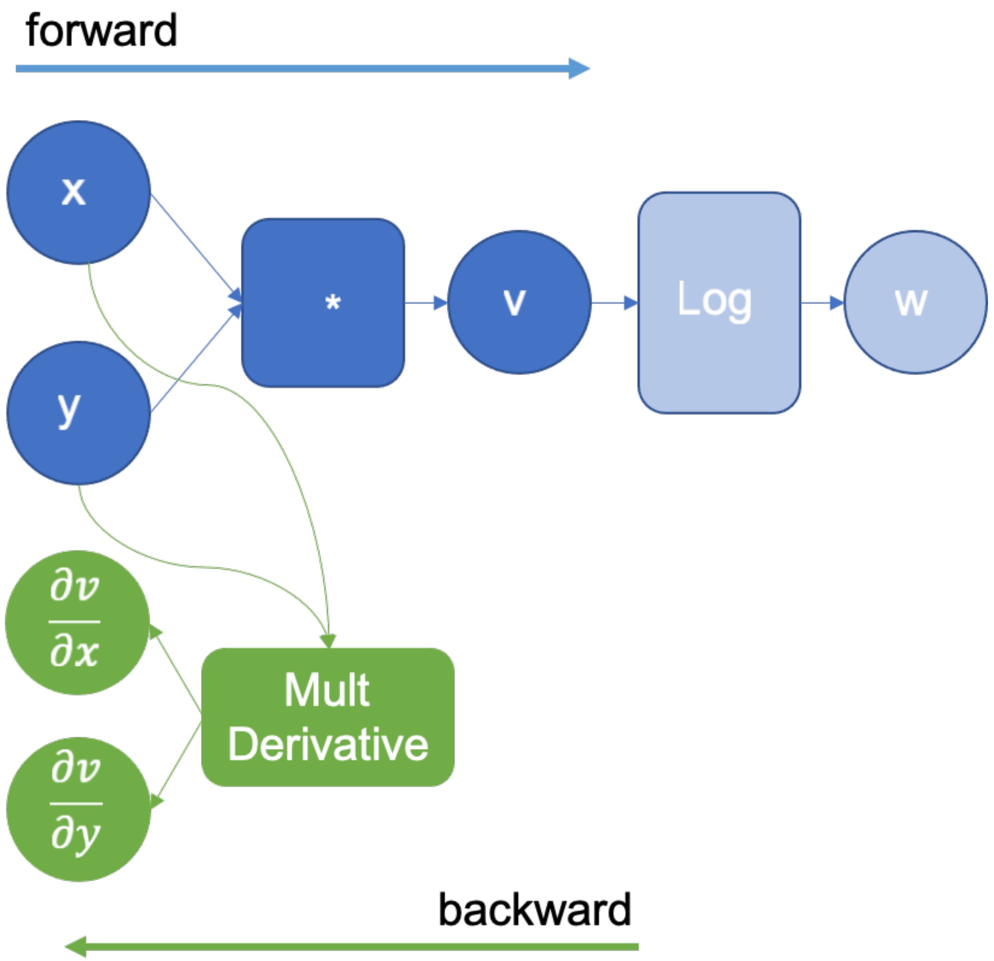


Image Credit: [6]

Computational Graphs & Autograd

If you need to compute gradients in the computational graph, you need to set the `requires_grad` attribute to be true for the tensor.

$$f(x, y) = \log(xy)$$

Every time Autograd (i.e., the automatic differentiation engine) executes an operation in the graph, the derivative of that operation is added to the graph to be executed later in the backward pass. Note that Autograd knows the derivatives of the basic functions.

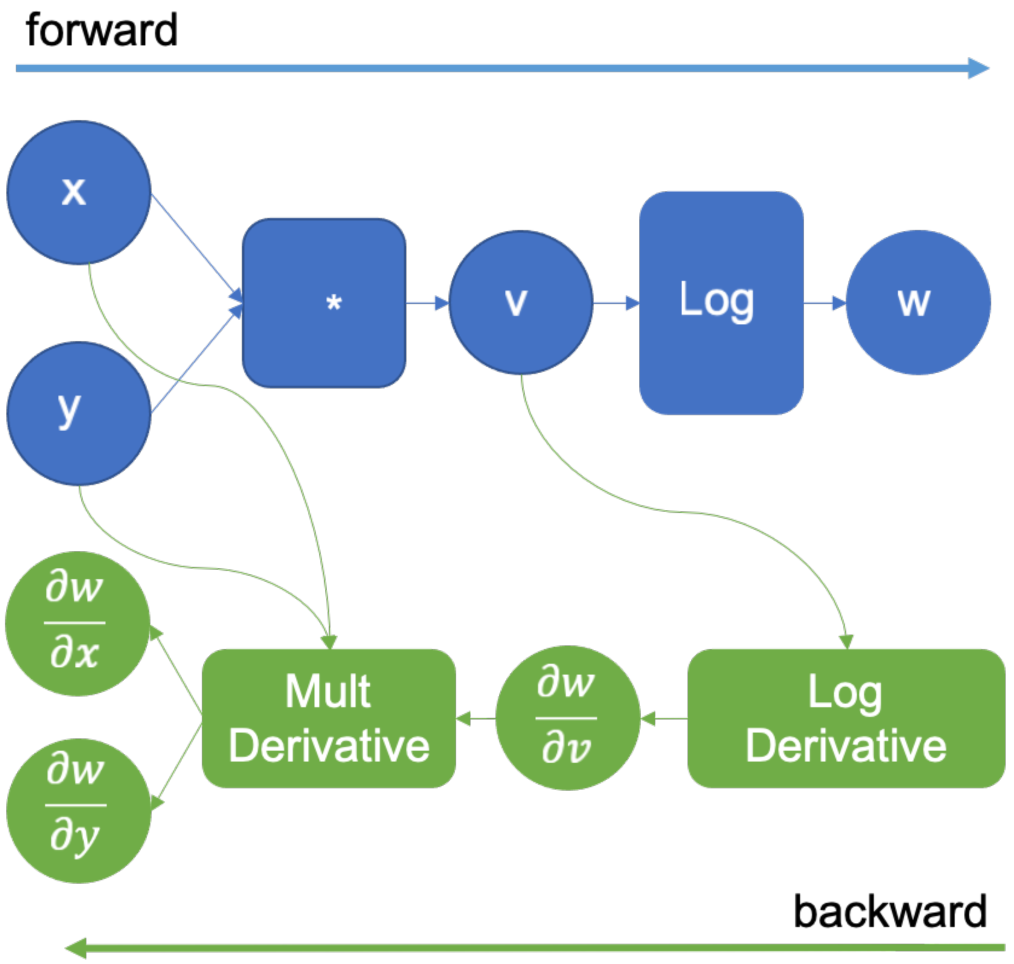


Image Credit: [6]

Computational Graphs & Autograd

If you need to compute gradients in the computational graph, you need to set the `requires_grad` attribute to be true for the tensor.

$$f(x, y) = \log(xy)$$

Every time Autograd (i.e., the automatic differentiation engine) executes an operation in the graph, the derivative of that operation is added to the graph to be executed later in the backward pass. Note that Autograd knows the derivatives of the basic functions.

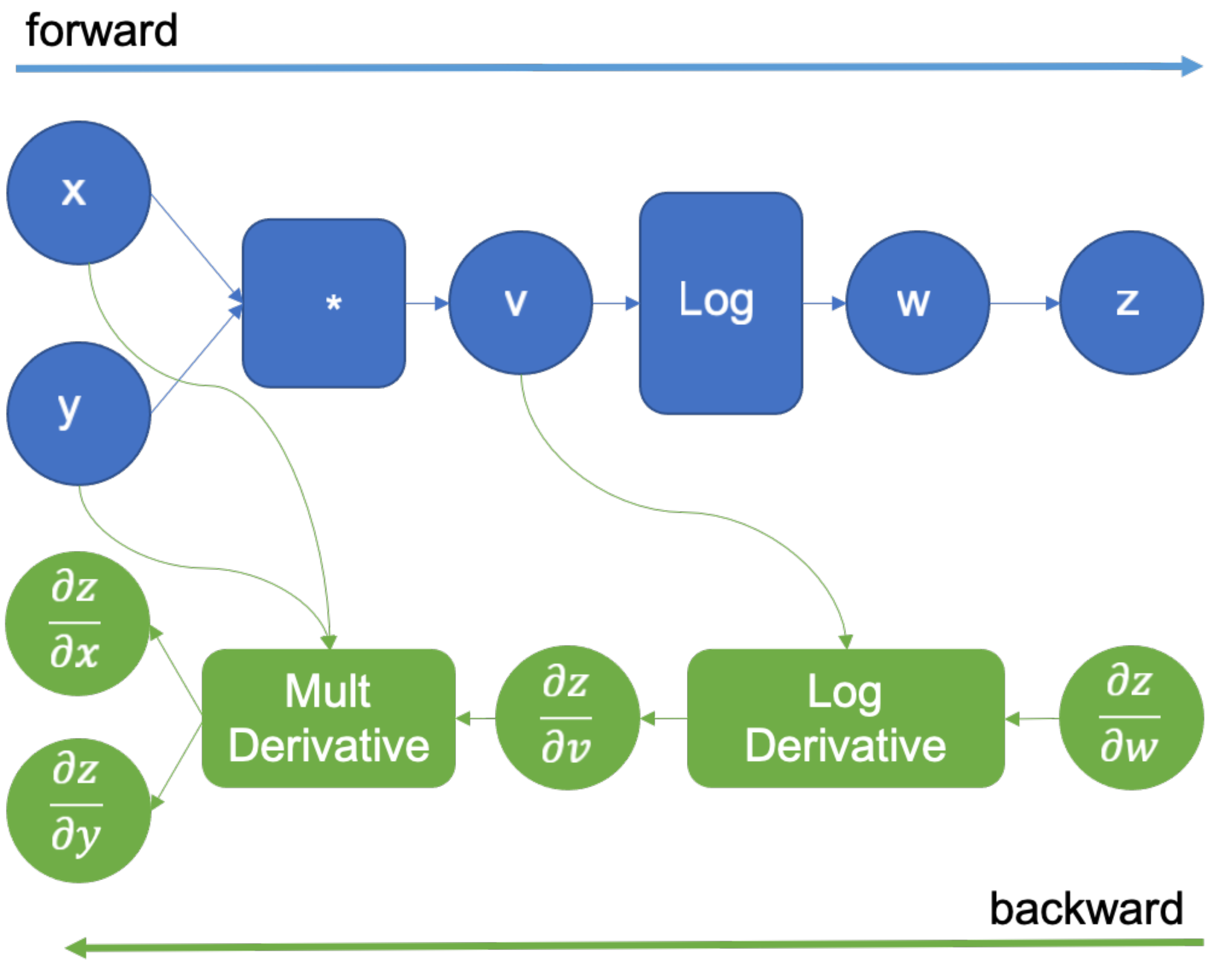


Image Credit: [6]

Computational Graphs & Autograd

If you need to compute gradients in the computational graph, you need to set the `requires_grad` attribute to be true for the tensor.

$$f(x, y) = \log(xy)$$

Every time Autograd (i.e., the automatic differentiation engine) executes an operation in the graph, the derivative of that operation is added to the graph to be executed later in the backward pass. Note that Autograd knows the derivatives of the basic functions.

Recall as we learn in BP, Autograd always computes scalar-by-xxx gradients via *Jacobian (transposed) vector product* and never explicitly forms a Jacobian matrix!

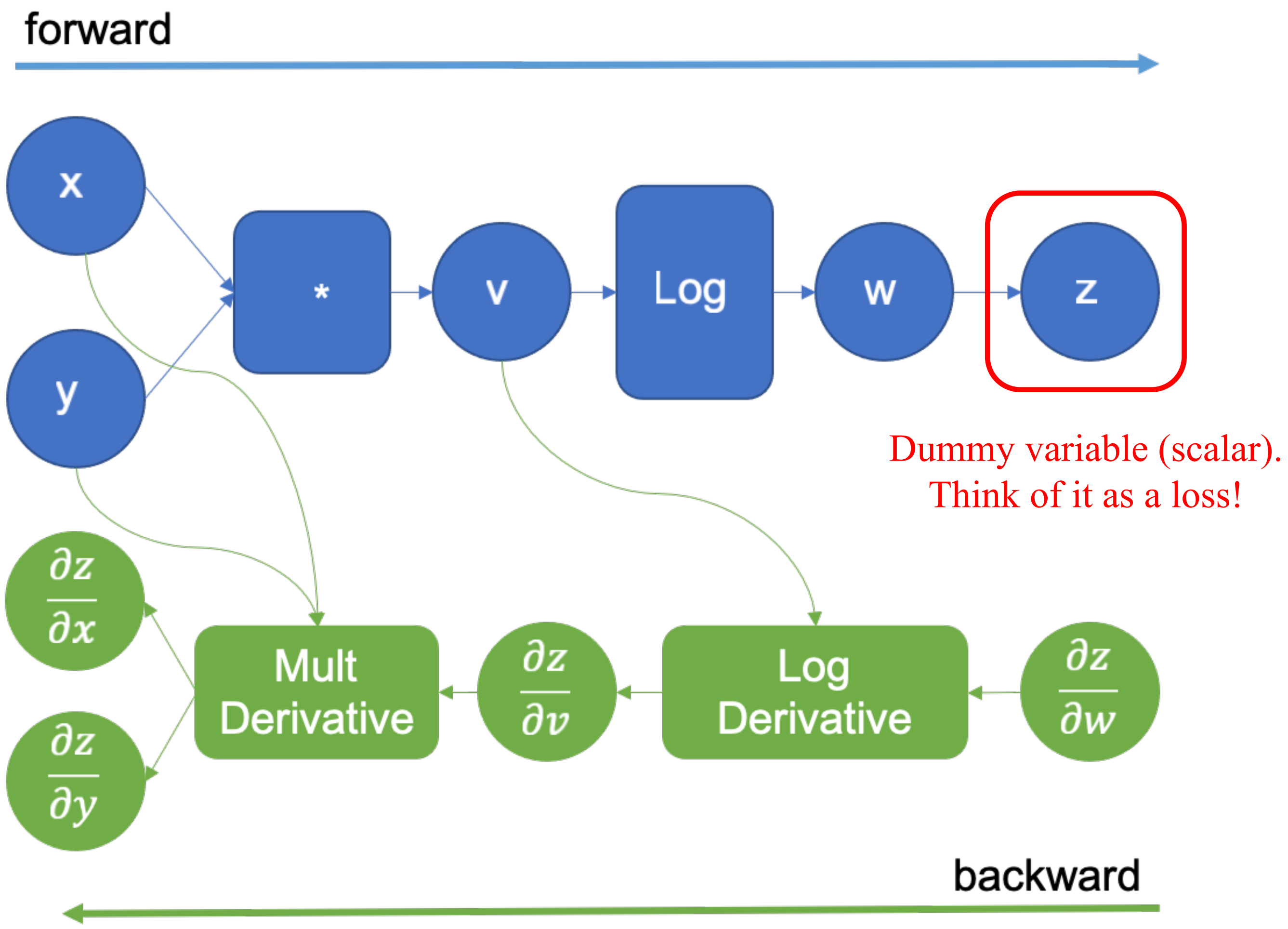


Image Credit: [6]

Computational Graphs & Autograd

If you need to compute gradients in the computational graph, you need to set the `requires_grad` attribute to be true for the tensor.

$$f(x, y) = \log(xy)$$

```
In [1]: import torch

In [2]: x = torch.tensor(0.5, requires_grad=True)

In [3]: y = torch.tensor(0.75)

In [4]: w = torch.log(x * y)

In [5]: w
Out[5]: tensor(-0.9808, grad_fn=<LogBackward>)

In [6]: grad_x = torch.autograd.grad(w, x)

In [7]: grad_x
Out[7]: (tensor(2.),)
```

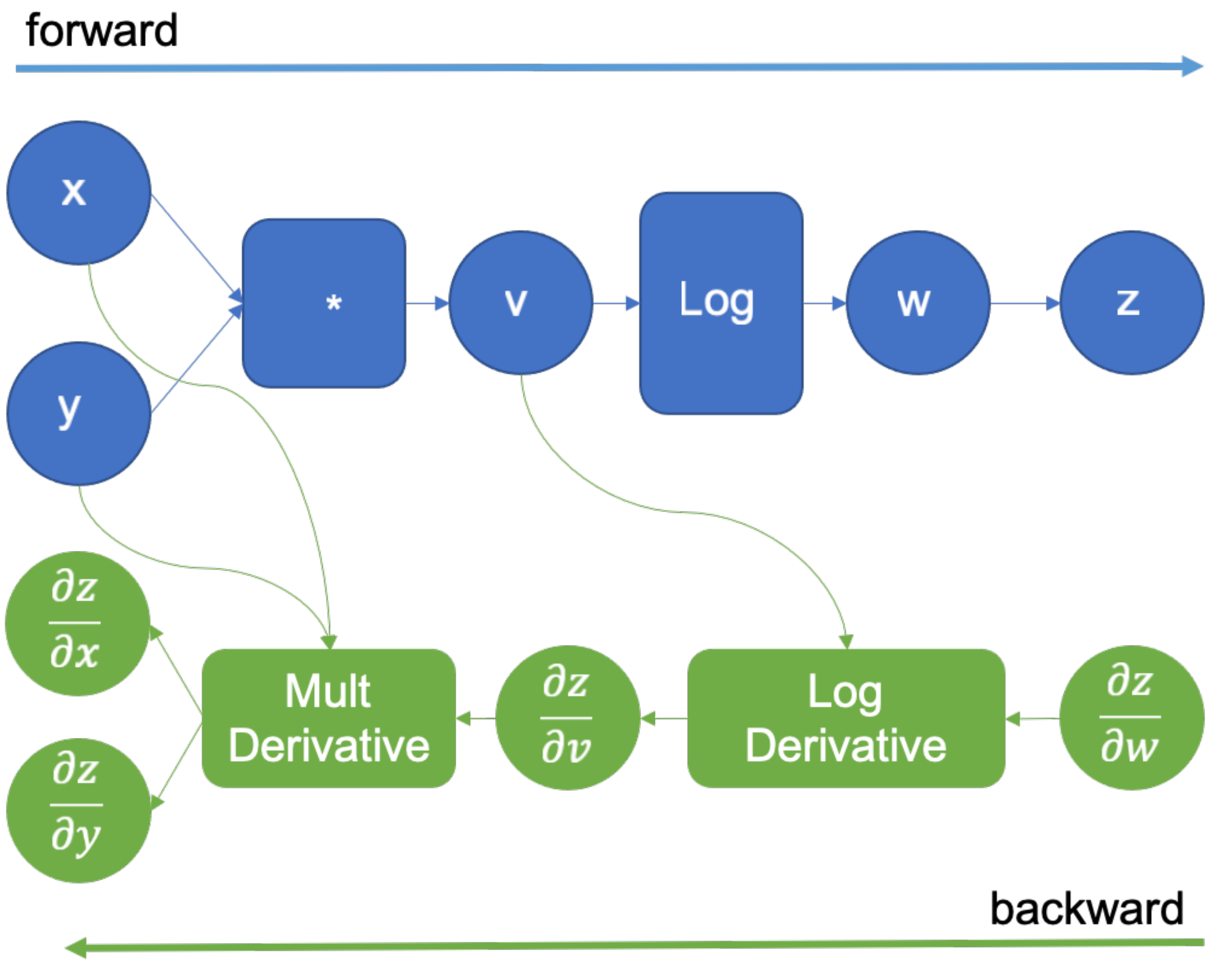


Image Credit: [6]

Computational Graphs & Autograd

If you need to compute gradients in the computational graph, you need to set the `requires_grad` attribute to be true for the tensor.

$$f(x, y) = \log(xy)$$

```
In [1]: import torch

In [2]: x = torch.tensor(0.5, requires_grad=True)

In [3]: y = torch.tensor(0.75)

In [4]: w = torch.log(x * y)

In [5]: w
Out[5]: tensor(-0.9808, grad_fn=<LogBackward>)

In [6]: grad_x = torch.autograd.grad(w, x)

In [7]: grad_x
Out[7]: (tensor(2.),)
```

More details of how computational graphs are constructed and executed can be found in [7,8].

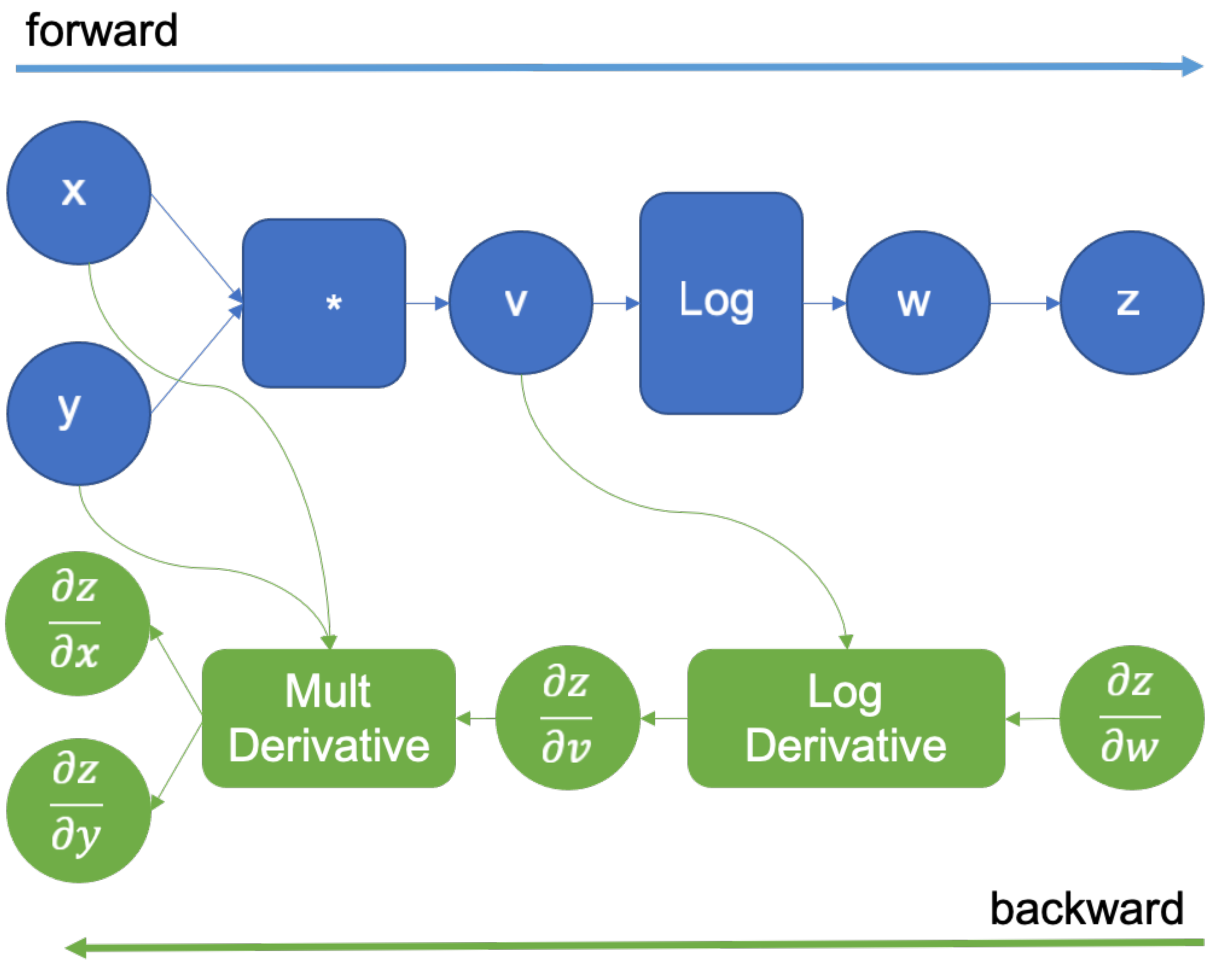


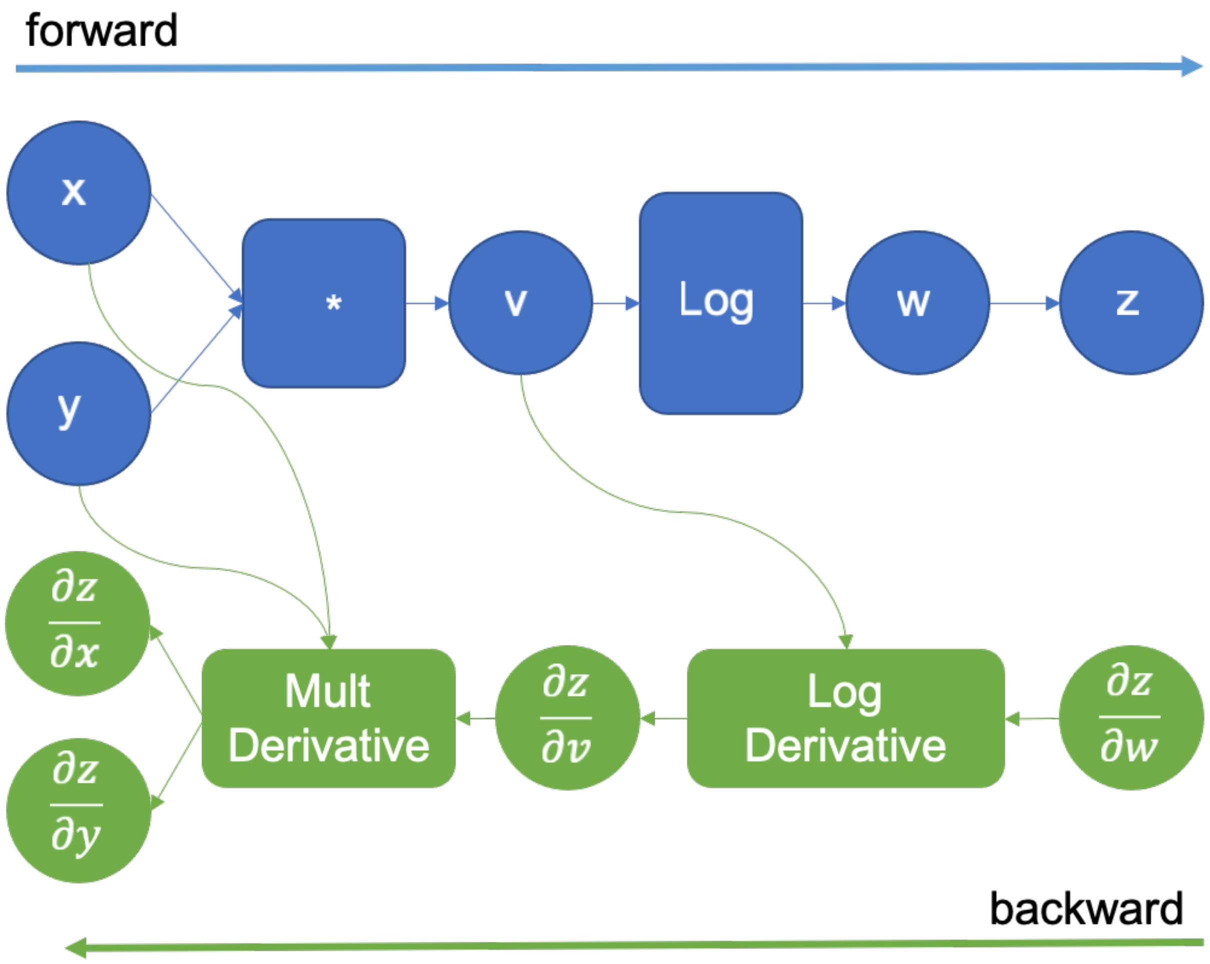
Image Credit: [6]

Computational Graphs & Autograd

If you need to compute gradients in the computational graph, you need to set the `requires_grad` attribute to be true for the tensor.

$$f(x, y) = \log(xy)$$

```
In [1]: import torch

In [2]: x = torch.tensor(0.5, requires_grad=True)

In [3]: y = torch.tensor(0.75)

In [4]: w = torch.log(x * y)

In [5]: w
Out[5]: tensor(-0.9808, grad_fn=<LogBackward>)

In [6]: grad_x = torch.autograd.grad(w, x)

In [7]: grad_x
Out[7]: (tensor(2.),)
```



More details of how computational graphs are constructed and executed can be found in [7,8].

Image Credit: [6]

Gradient Checking

How can we check if gradients (even those returned by Autograd) are correctly implemented?

Gradient Checking

How can we check if gradients (even those returned by Autograd) are correctly implemented?

Consider $y = f(\mathbf{x})$ where $\mathbf{x} \in \mathbb{R}^d$ $y \in \mathbb{R}$

Gradient Checking

How can we check if gradients (even those returned by Autograd) are correctly implemented?

Consider $y = f(\mathbf{x})$ where $\mathbf{x} \in \mathbb{R}^d$ $y \in \mathbb{R}$

Recall $\nabla f(\mathbf{p}) = \begin{bmatrix} \frac{\partial f}{\partial \mathbf{x}_1}(\mathbf{p}) \\ \vdots \\ \frac{\partial f}{\partial \mathbf{x}_d}(\mathbf{p}) \end{bmatrix}$ and $\frac{\partial f}{\partial \mathbf{x}_i}(\mathbf{p}) = \lim_{\epsilon \to 0} \frac{f(\mathbf{p} + \epsilon \mathbf{e}_i) - f(\mathbf{p})}{\epsilon}$

Here we use the standard basis vector $\mathbf{e}_i = [0, \cdots 0, 1, 0, \cdots 0]$

i-th entry

Gradient Checking

How can we check if gradients (even those returned by Autograd) are correctly implemented?

Consider $y = f(\mathbf{x})$ where $\mathbf{x} \in \mathbb{R}^d$ $y \in \mathbb{R}$

Recall $\nabla f(\mathbf{p}) = \begin{bmatrix} \frac{\partial f}{\partial \mathbf{x}_1}(\mathbf{p}) \\ \vdots \\ \frac{\partial f}{\partial \mathbf{x}_d}(\mathbf{p}) \end{bmatrix}$ and $\frac{\partial f}{\partial \mathbf{x}_i}(\mathbf{p}) = \lim_{\epsilon \to 0} \frac{f(\mathbf{p} + \epsilon \mathbf{e}_i) - f(\mathbf{p})}{\epsilon}$

Here we use the standard basis vector $\mathbf{e}_i = [0, \cdots 0, 1, 0, \cdots 0]$

i-th entry

Based on the (forward difference) finite approximation, we have

$$\frac{\partial f}{\partial \mathbf{x}_i}(\mathbf{p}) = \lim_{\epsilon \to 0} \frac{f(\mathbf{p} + \epsilon \mathbf{e}_i) - f(\mathbf{p})}{\epsilon} \approx \frac{f(\mathbf{p} + \epsilon \mathbf{e}_i) - f(\mathbf{p})}{\epsilon}$$

One can also use central difference finite approximation. Make sure using float64 with small perturbation, e.g., $\epsilon = 1e^{-5}$

Gradient Checking

How can we check if gradients (even those returned by Autograd) are correctly implemented?

Consider $y = f(\mathbf{x})$ where $\mathbf{x} \in \mathbb{R}^d$ $y \in \mathbb{R}$

Recall $\nabla f(\mathbf{p}) = \begin{bmatrix} \frac{\partial f}{\partial \mathbf{x}_1}(\mathbf{p}) \\ \vdots \\ \frac{\partial f}{\partial \mathbf{x}_d}(\mathbf{p}) \end{bmatrix}$ and $\frac{\partial f}{\partial \mathbf{x}_i}(\mathbf{p}) = \lim_{\epsilon \to 0} \frac{f(\mathbf{p} + \epsilon \mathbf{e}_i) - f(\mathbf{p})}{\epsilon}$

Here we use the standard basis vector $\mathbf{e}_i = [0, \cdots 0, \underset{\uparrow}{1}, 0, \cdots 0]$

i-th entry

Based on the (forward difference) finite approximation, we have

$$\frac{\partial f}{\partial \mathbf{x}_i}(\mathbf{p}) = \lim_{\epsilon \to 0} \frac{f(\mathbf{p} + \epsilon \mathbf{e}_i) - f(\mathbf{p})}{\epsilon} \approx \frac{f(\mathbf{p} + \epsilon \mathbf{e}_i) - f(\mathbf{p})}{\epsilon}$$

One can also use central difference finite approximation. Make sure using float64 with small perturbation, e.g., $\epsilon = 1e^{-5}$

How many times of function evaluations do we need to compute the gradient in this case?

Outline

- Basics in PyTorch
- Computational Graphs, Autograd, and Gradient Checking
- **Creating Models**
- Loading Data and Training Models

Creating Models

We can create a MLP with two hidden layers in PyTorch as follows

```python
import torch
from torch import nn # Import the nn sub-module from PyTorch

class NeuralNetwork(nn.Module): # Neural networks are defined as classes
    def __init__(self): # Layers and variables are defined in the __init__ method
        super(NeuralNetwork, self).__init__() # Must be in every network.
        self.flatten = nn.Flatten() # Defining a flattening layer.
        self.linear_relu_stack = nn.Sequential( # Defining a stack of layers.
            nn.Linear(28*28, 512), # Linear Layers have an input and output shape
            nn.ReLU(), # ReLU is one of many activation functions provided by nn
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )

    def forward(self, x): # This function defines the forward pass.
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```


Image Credit: [5]

Creating Models

We can create a MLP with two hidden layers in PyTorch as follows

Specify the model

```python
import torch
from torch import nn # Import the nn sub-module from PyTorch

class NeuralNetwork(nn.Module): # Neural networks are defined as classes
    def __init__(self): # Layers and variables are defined in the __init__ method
        super(NeuralNetwork, self).__init__() # Must be in every network.
        self.flatten = nn.Flatten() # Defining a flattening layer.
        self.linear_relu_stack = nn.Sequential( # Defining a stack of layers.
            nn.Linear(28*28, 512), # Linear Layers have an input and output shape
            nn.ReLU(), # ReLU is one of many activation functions provided by nn
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )

    def forward(self, x): # This function defines the forward pass.
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```


Image Credit: [5]

Creating Models

We can create a MLP with two hidden layers in PyTorch as follows

Specify the model

```python
import torch
from torch import nn # Import the nn sub-module from PyTorch

class NeuralNetwork(nn.Module): # Neural networks are defined as classes
    def __init__(self): # Layers and variables are defined in the __init__ method
        super(NeuralNetwork, self).__init__() # Must be in every network.
        self.flatten = nn.Flatten() # Defining a flattening layer.
        self.linear_relu_stack = nn.Sequential( # Defining a stack of layers.
            nn.Linear(28*28, 512), # Linear Layers have an input and output shape
            nn.ReLU(), # ReLU is one of many activation functions provided by nn
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )

    def forward(self, x): # This function defines the forward pass.
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

Execute the forward function will create the computational graph. Since parameters in `nn.Linear` have `requires_grad=True` by default, it will also create the backward graph for BP.

Image Credit: [5]

Creating Models

Let us zoom in to the `nn.Linear`:

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

def reset_parameters(self) -> None:
    # Setting a=sqrt(5) in kaiming_uniform is the same as initializing with
    # uniform(-1/sqrt(in_features), 1/sqrt(in_features)). For details, see
    # https://github.com/pytorch/pytorch/issues/57109
    init.kaiming_uniform_(self.weight, a=math.sqrt(5))
    if self.bias is not None:
        fan_in, _ = init._calculate_fan_in_and_fan_out(self.weight)
        bound = 1 / math.sqrt(fan_in) if fan_in > 0 else 0
        init.uniform_(self.bias, -bound, bound)

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


Image Credit: [9]

Creating Models

Let us zoom in to the `nn.Linear`:

`nn.Parameter` will create tensors of parameters which by default require gradients

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

def reset_parameters(self) -> None:
    # Setting a=sqrt(5) in kaiming_uniform is the same as initializing with
    # uniform(-1/sqrt(in_features), 1/sqrt(in_features)). For details, see
    # https://github.com/pytorch/pytorch/issues/57109
    init.kaiming_uniform_(self.weight, a=math.sqrt(5))
    if self.bias is not None:
        fan_in, _ = init._calculate_fan_in_and_fan_out(self.weight)
        bound = 1 / math.sqrt(fan_in) if fan_in > 0 else 0
        init.uniform_(self.bias, -bound, bound)

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


Image Credit: [9]

Creating Models

Let us zoom in to the `nn.Linear`:

`nn.Parameter` will create tensors of parameters which by default require gradients

Initialization of parameters

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

def reset_parameters(self) -> None:
    # Setting a=sqrt(5) in kaiming_uniform is the same as initializing with
    # uniform(-1/sqrt(in_features), 1/sqrt(in_features)). For details, see
    # https://github.com/pytorch/pytorch/issues/57109
    init.kaiming_uniform_(self.weight, a=math.sqrt(5))
    if self.bias is not None:
        fan_in, _ = init._calculate_fan_in_and_fan_out(self.weight)
        bound = 1 / math.sqrt(fan_in) if fan_in > 0 else 0
        init.uniform_(self.bias, -bound, bound)

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


Image Credit: [9]

Creating Models

Let us zoom in to the `nn.Linear`:

`nn.Parameter` will create tensors of parameters which by default require gradients

Initialization of parameters

Computation in forward pass

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

def reset_parameters(self) -> None:
    # Setting a=sqrt(5) in kaiming_uniform is the same as initializing with
    # uniform(-1/sqrt(in_features), 1/sqrt(in_features)). For details, see
    # https://github.com/pytorch/pytorch/issues/57109
    init.kaiming_uniform_(self.weight, a=math.sqrt(5))
    if self.bias is not None:
        fan_in, _ = init._calculate_fan_in_and_fan_out(self.weight)
        bound = 1 / math.sqrt(fan_in) if fan_in > 0 else 0
        init.uniform_(self.bias, -bound, bound)

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


Image Credit: [9]

Outline

- Basics in PyTorch
- Computational Graphs, Autograd, and Gradient Checking
- Creating A Model
- **Loading Data and Training Models**

Load Data & Training Models

Let us load the Fashion MNIST dataset (many public datasets are available in `torchvision`) and train the previous MLP:

```
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor

training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor()
)

test_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor()
)

train_dataloader = DataLoader(training_data, batch_size=64)
test_dataloader = DataLoader(test_data, batch_size=64)
```

Image Credit: [10]

Load Data & Training Models

Let us load the Fashion MNIST dataset (many public datasets are available in `torchvision`) and train the previous MLP:

Transform images (e.g., PNG) to PyTorch tensors. You can check `torchvision.transforms` for more transformations!

```
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor

training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor()
)

test_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor()
)

train_dataloader = DataLoader(training_data, batch_size=64)
test_dataloader = DataLoader(test_data, batch_size=64)
```

Image Credit: [10]

Load Data & Training Models

Let us load the Fashion MNIST dataset (many public datasets are available in `torchvision`) and train the previous MLP:

Sampled images from Fashion MNIST [14]

```
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor

training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor()
)

test_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor()
)

train_dataloader = DataLoader(training_data, batch_size=64)
test_dataloader = DataLoader(test_data, batch_size=64)
```

Image Credit: [10]

Load Data & Training Models

You can also customize your dataloader:

```
import os
import pandas as pd
from torchvision.io import read_image

class CustomImageDataset(Dataset):
    def __init__(self, annotations_file, img_dir, transform=None, target_transform=None):
        self.img_labels = pd.read_csv(annotations_file)
        self.img_dir = img_dir
        self.transform = transform
        self.target_transform = target_transform

    def __len__(self):
        return len(self.img_labels)

    def __getitem__(self, idx):
        img_path = os.path.join(self.img_dir, self.img_labels.iloc[idx, 0])
        image = read_image(img_path)
        label = self.img_labels.iloc[idx, 1]
        if self.transform:
            image = self.transform(image)
        if self.target_transform:
            label = self.target_transform(label)
        return image, label
```


Image Credit: [10]

Load Data & Training Models

You can also customize your dataloader:

```
import os
import pandas as pd
from torchvision.io import read_image

class CustomImageDataset(Dataset):
    def __init__(self, annotations_file, img_dir, transform=None, target_transform=None):
        self.img_labels = pd.read_csv(annotations_file)
        self.img_dir = img_dir
        self.transform = transform
        self.target_transform = target_transform

    def __len__(self):
        return len(self.img_labels)

    def __getitem__(self, idx):
        img_path = os.path.join(self.img_dir, self.img_labels.iloc[idx, 0])
        image = read_image(img_path)
        label = self.img_labels.iloc[idx, 1]
        if self.transform:
            image = self.transform(image)
        if self.target_transform:
            label = self.target_transform(label)
        return image, label
```

You can override the function `__getitem__` which extracts a single data example within a mini-batch!

Image Credit: [10]

Load Data & Training Models

You can also customize your dataloader:

PyTorch dataloaders collate individual fetched data samples into a mini-batch via `collate_fn` function which can be customized as well. See [11,12] for more details.

You can override the function `__getitem__` which extracts a single data example within a mini-batch!

```
import os
import pandas as pd
from torchvision.io import read_image

class CustomImageDataset(Dataset):
    def __init__(self, annotations_file, img_dir, transform=None, target_transform=None):
        self.img_labels = pd.read_csv(annotations_file)
        self.img_dir = img_dir
        self.transform = transform
        self.target_transform = target_transform

    def __len__(self):
        return len(self.img_labels)

    def __getitem__(self, idx):
        img_path = os.path.join(self.img_dir, self.img_labels.iloc[idx, 0])
        image = read_image(img_path)
        label = self.img_labels.iloc[idx, 1]
        if self.transform:
            image = self.transform(image)
        if self.target_transform:
            label = self.target_transform(label)
        return image, label
```


Image Credit: [10]

Load Data & Training Models

Now let us see how we can load data and train models:

```python
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")

loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
                                    torch.nn.Module.parameters
epochs = 10
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train_loop(train_dataloader, model, loss_fn, optimizer)
    test_loop(test_dataloader, model, loss_fn)
print("Done!")
```


Image Credit: [13]

Load Data & Training Models

Now let us see how we can load data and train models:

Loop over all mini-batches within the dataset

```python
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")

loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
                                torch.nn.Module.parameters

epochs = 10
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train_loop(train_dataloader, model, loss_fn, optimizer)
    test_loop(test_dataloader, model, loss_fn)
print("Done!")
```


Image Credit: [13]

Load Data & Training Models

Now let us see how we can load data and train models:

Compute forward pass & loss

```
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")

loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

torch.nn.Module.parameters

```
epochs = 10
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train_loop(train_dataloader, model, loss_fn, optimizer)
    test_loop(test_dataloader, model, loss_fn)
print("Done!")
```


Image Credit: [13]

Load Data & Training Models

Now let us see how we can load data and train models:

Clean cached gradients from previous mini-batches

```
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")

loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
                                  torch.nn.Module.parameters

epochs = 10
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train_loop(train_dataloader, model, loss_fn, optimizer)
    test_loop(test_dataloader, model, loss_fn)
print("Done!")
```


Image Credit: [13]

Load Data & Training Models

Now let us see how we can load data and train models:

Compute gradient via backpropagation

```python
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")

loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

torch.nn.Module.parameters

```python
epochs = 10
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train_loop(train_dataloader, model, loss_fn, optimizer)
    test_loop(test_dataloader, model, loss_fn)
print("Done!")
```

Image Credit: [13]

Load Data & Training Models

Now let us see how we can load data and train models:

Update parameters via the optimizer (e.g., SGD/Adam)

```
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")

loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
                                      torch.nn.Module.parameters
epochs = 10
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train_loop(train_dataloader, model, loss_fn, optimizer)
    test_loop(test_dataloader, model, loss_fn)
print("Done!")
```


Image Credit: [13]

Load Data & Training Models

Now let us see how we can load data and train models:

Specify loss function and optimizer

```
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")

loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
                                torch.nn.Module.parameters

epochs = 10
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train_loop(train_dataloader, model, loss_fn, optimizer)
    test_loop(test_dataloader, model, loss_fn)
print("Done!")
```

Image Credit: [13]

Load Data & Training Models

Now let us see how we can load data and train models:

```
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")

loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
                                   torch.nn.Module.parameters

epochs = 10
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train_loop(train_dataloader, model, loss_fn, optimizer)
    test_loop(test_dataloader, model, loss_fn)
print("Done!")
```

One call of `train_loop` amounts to training for one epoch

Image Credit: [13]

Load Data & Training Models

Test loop is similar to train loop:

```
def test_loop(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    test_loss, correct = 0, 0

    with torch.no_grad():
        for X, y in dataloader:
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()

    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_loss:>8f} \n")
```


Image Credit: [13]

Load Data & Training Models

Test loop is similar to train loop:

```
def test_loop(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    test_loss, correct = 0, 0

    with torch.no_grad():
        for X, y in dataloader:
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()

    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_loss:>8f} \n")
```

We do not need to create the backward part of the computational graph. Call `torch.no_grad()` could save us some GPU memory!

Image Credit: [13]

Load Data & Training Models

Test loop is similar to train loop:

```
def test_loop(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    test_loss, correct = 0, 0

    with torch.no_grad():
        for X, y in dataloader:
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()

    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_loss:>8f} \n")
```

Counting the number of correctly classified samples.

Image Credit: [13]

References

[1] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., ... & Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. Advances in neural information processing systems, 32.

[2] https://realpython.com/pytorch-vs-tensorflow/

[3] https://en.wikipedia.org/wiki/Comparison_of_deep_learning_software

[4] https://www.reddit.com/r/MachineLearning/comments/rga91a/d_are_you_using_pytorch_or_tensorflow_going_into/

[5] https://en.wikipedia.org/wiki/PyTorch

[6] https://pytorch.org/blog/overview-of-pytorch-autograd-engine/

[7] https://pytorch.org/blog/computational-graphs-constructed-in-pytorch/

[8] https://pytorch.org/blog/how-computational-graphs-are-executed-in-pytorch/

[9] https://pytorch.org/docs/stable/_modules/torch/nn/modules/linear.html#Linear

[10] https://pytorch.org/tutorials/beginner/basics/data_tutorial.html

[11] https://pytorch.org/tutorials/beginner/data_loading_tutorial.html

[12] https://pytorch.org/docs/stable/data.html#loading-batched-and-non-batched-data

[13] https://pytorch.org/tutorials/beginner/basics/optimization_tutorial.html

[14] Xiao, H., Rasul, K., & Vollgraf, R. (2017). Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. arXiv preprint arXiv:1708.07747.

Questions?