Introduction to Pytorch

CPEN 455 Tutorial 2

Muchen Li

Last Time

- Tensor operations
  - Index, Slice, Stride, Concatenate, reshape
- Dataset and DataLoader Module

This Time

- Autograd
- Build Model
- Model Optimization
  - Learning Rate
  - Optimizer
  - Hyper parameter choice

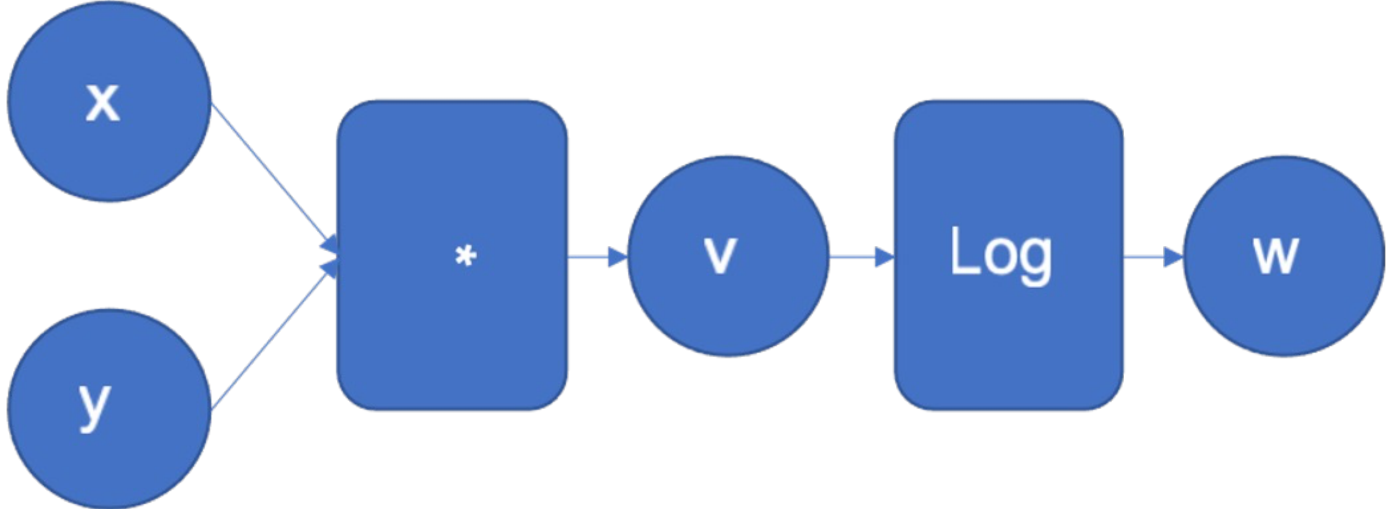
3. Autograd

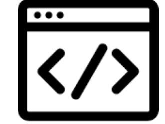
- When training neural networks, the most frequently used algorithm is **back propagation**. In this algorithm, parameters (model weights) are adjusted according to the **gradient** of the loss function with respect to the given parameter.

- To compute those gradients, PyTorch has a built-in differentiation engine called torch.autograd. It supports automatic computation of gradient for any computational graph.

3. Autograd: Computation Graph

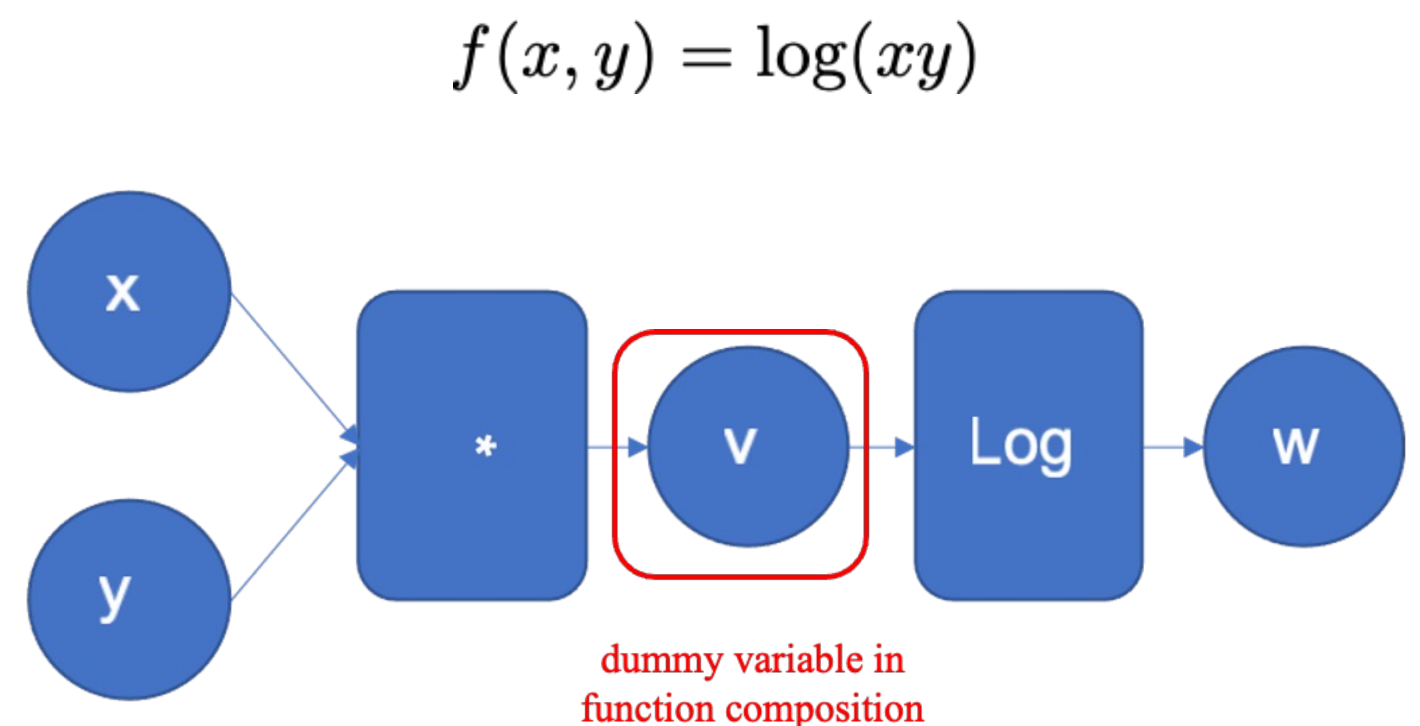
$$f(x, y) = \log(xy)$$



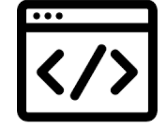


[1] Figure Credit: https://pytorch.org/blog/overview-of-pytorch-autograd-engine/

3. Autograd: Computation Graph



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).



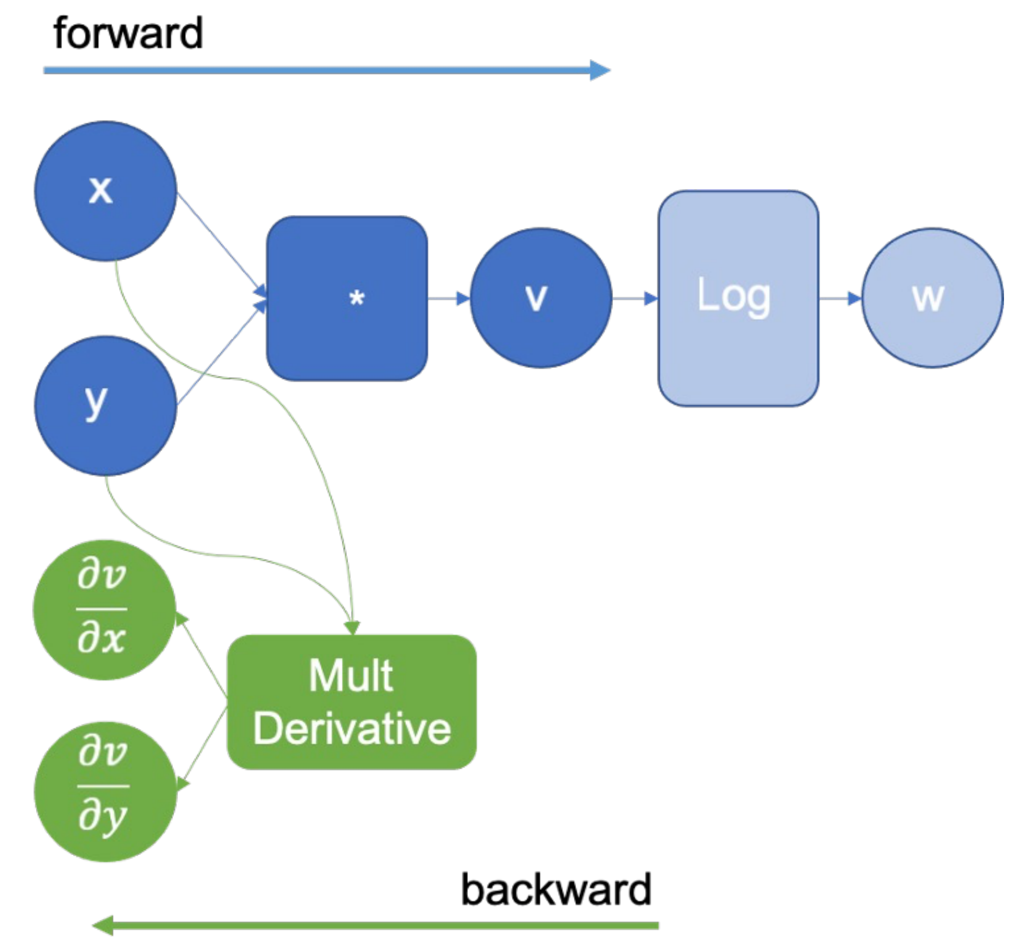
[1] Figure Credit: https://pytorch.org/blog/overview-of-pytorch-autograd-engine/

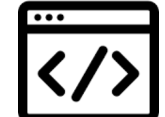
3. Autograd

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

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.





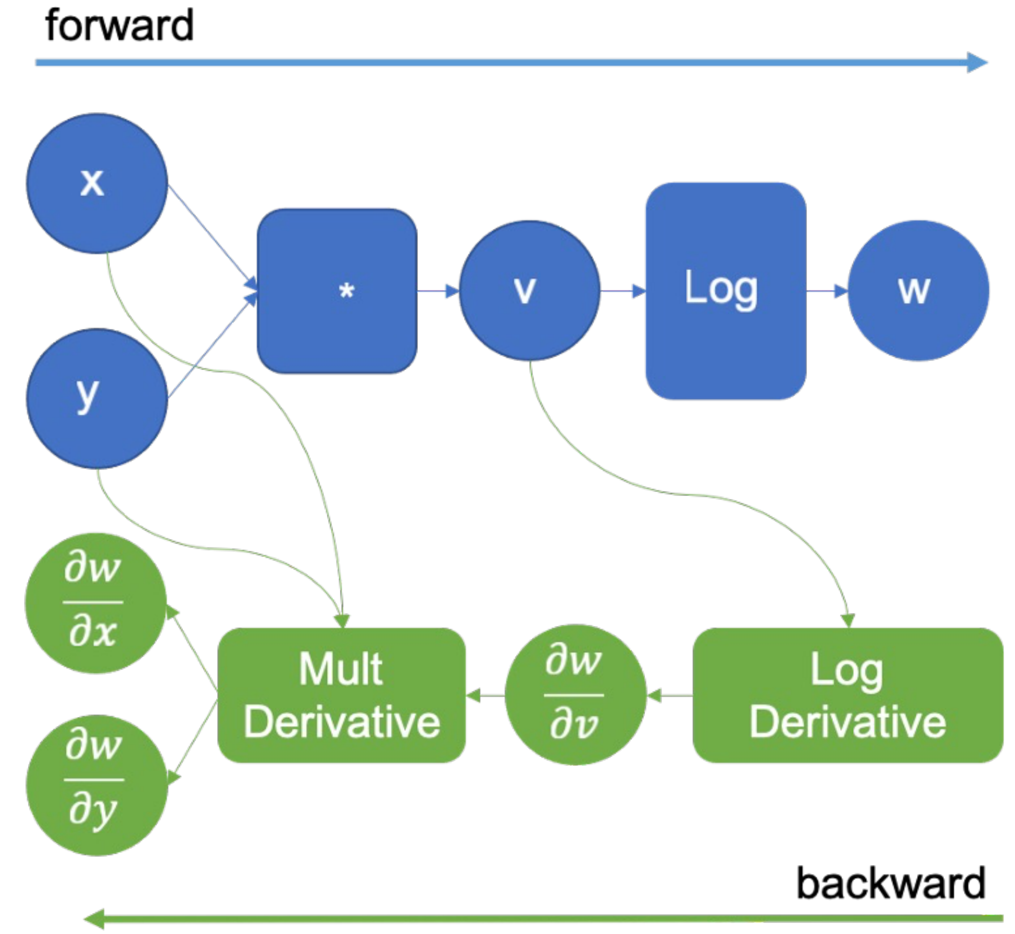
[1] Figure Credit: https://pytorch.org/blog/overview-of-pytorch-autograd-engine/

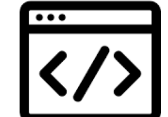
3. Autograd

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

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.





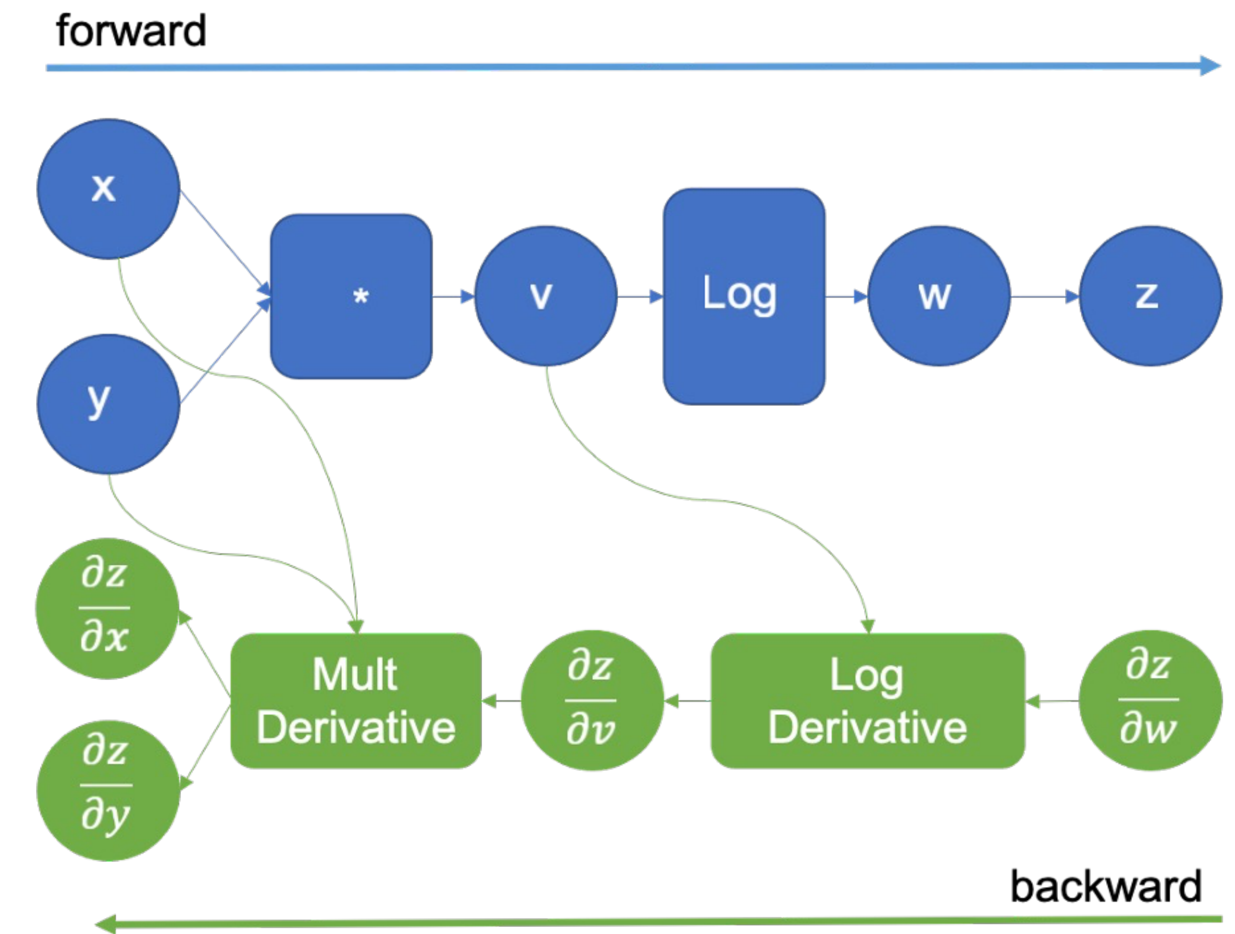
[1] Figure Credit: https://pytorch.org/blog/overview-of-pytorch-autograd-engine/

3. Autograd

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

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.



[1] Figure Credit: https://pytorch.org/blog/overview-of-pytorch-autograd-engine/

3. Autograd

For a vector function $\vec{y} = f(\vec{x})$

Where $\vec{x} = \langle x_1, \ldots, x_n \rangle$ and $\vec{y} = \langle y_1, \ldots, y_m \rangle$

A gradient of y with respect to x is given by **Jacobian matrix**:

3. Autograd

For a vector function $\vec{y} = f(\vec{x})$

Where $\vec{x} = \langle x_1, \ldots, x_n \rangle$ and $\vec{y} = \langle y_1, \ldots, y_m \rangle$

A gradient of y with respect to x is given by **Jacobian matrix**:

$$J = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix}$$

3. Autograd

For a vector function $\vec{y} = f(\vec{x})$

Where $\vec{x} = \langle x_1, \ldots, x_n \rangle$ and $\vec{y} = \langle y_1, \ldots, y_m \rangle$

A gradient of y with respect to x is given by **Jacobian matrix**:

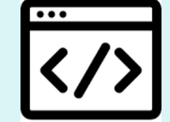
$$J = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix}$$

Instead of computing the Jacobian matrix itself, PyTorch allows you to compute Jacobian Product $v^T \cdot J$ $\quad v = (v_1 \ldots v_m)$

Tips 3.1 Disable Gradient Checking

- By default, all tensors with requires_grad=True are tracking their computational history and support gradient computation. However, there are some cases when we do not need to do that, for example, when we have trained the model and just want to apply it to some input data.
  - **torch.no_grad()**
  - **tensor.detach()**
  - **tensor.requires_grad = False**
- Let's look at some interesting example of the difference between these methods here

- There are reasons you might want to disable gradient tracking:
  - To mark some parameters in your neural network as **frozen parameters**.
  - To **speed up computations** when you are only doing forward pass, because computations on tensors that do not track gradients would be more efficient.



[2] https://pytorch.org/tutorials/beginner/basics/autogradqs_tutorial.html

Tips 3.2 Check your gradient

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

Tips 3.2 Check your gradient

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

Tips 3.2 Check your gradient

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

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

Tips 3.2 Check your gradient

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

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

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

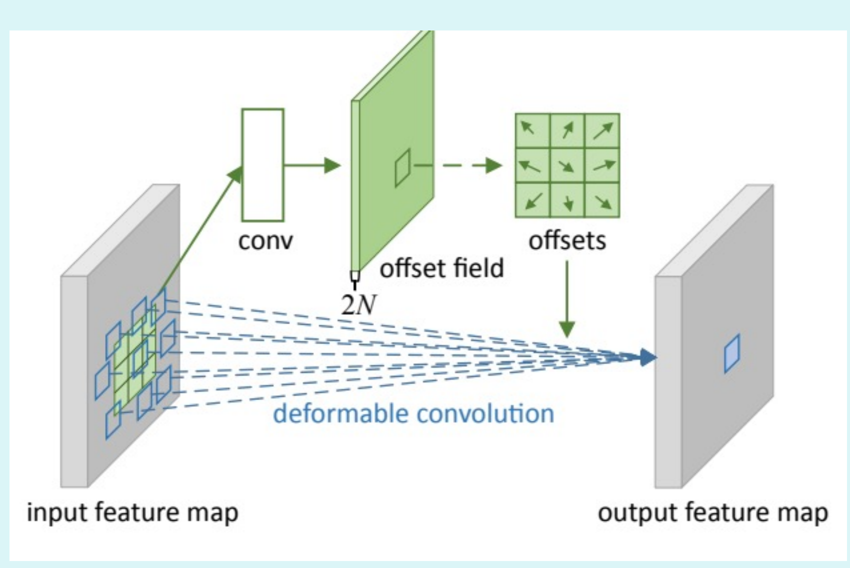
$$\frac{\partial f}{\partial \mathbf{x}_i}(\mathbf{p}) = \lim_{\epsilon \to 0} \frac{f(\mathbf{p} + \epsilon \mathbf{e}_i) - f(\mathbf{p})}{\epsilon} \approx \frac{f(\mathbf{p} + \epsilon \mathbf{e}_i) - f(\mathbf{p})}{\epsilon} \qquad \epsilon = 1e^{-5}$$

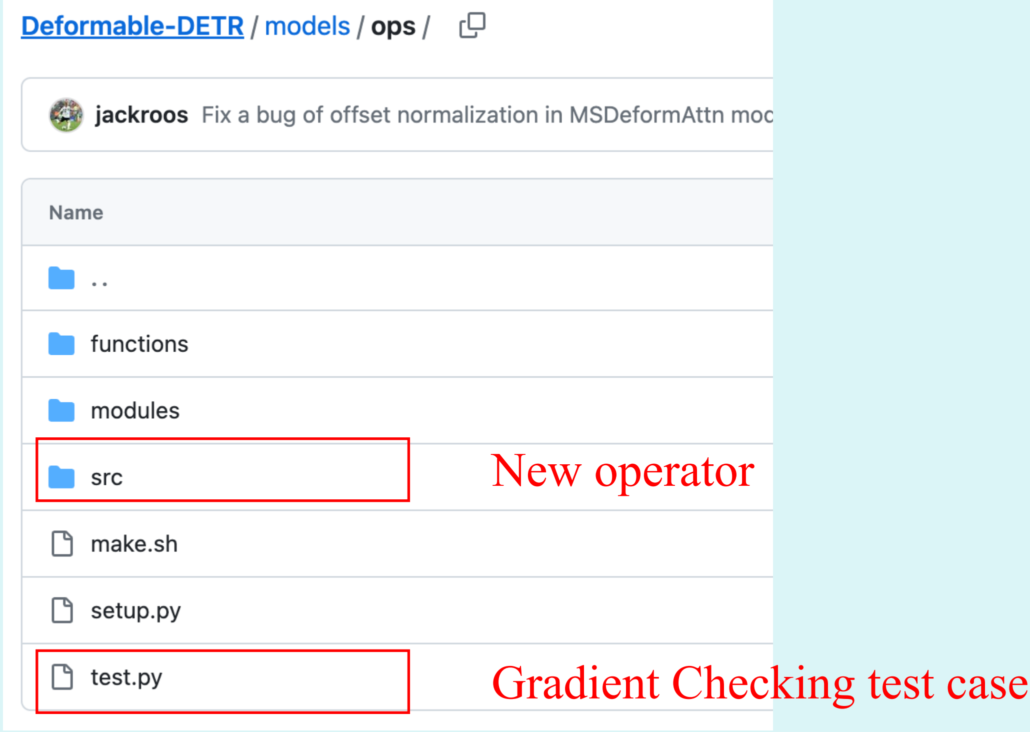
$$\mathbf{e}_i = [0, \cdots 0, 1, 0, \cdots 0]$$

i-th entry

Tips 3.2 Check your gradient

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





[7] Dai, Jifeng, et al. "Deformable convolutional networks." *Proceedings of the IEEE international conference on computer vision*. 2017.

[8] https://pytorch.org/docs/stable/generated/torch.autograd.gradcheck.html

Tips 3.2 Check your gradient

TORCH.AUTOGRAD.GRADCHECK

```
torch.autograd.gradcheck(func, inputs, *, eps=1e-06, atol=1e-05, rtol=0.001, raise_exception=True,
check_sparse_nnz=None, nondet_tol=0.0, check_undefined_grad=True, check_grad_dtypes=False,
check_batched_grad=False, check_batched_forward_grad=False, check_forward_ad=False,
check_backward_ad=True, fast_mode=False, masked=None)
```
[SOURCE]

Check gradients computed via small finite differences against analytical gradients wrt tensors in `inputs` that are of floating point or complex type and with `requires_grad=True`.

The check between numerical and analytical gradients uses `allclose()`.

For most of the complex functions we consider for optimization purposes, no notion of Jacobian exists. Instead, gradcheck verifies if the numerical and analytical values of the Wirtinger and Conjugate Wirtinger derivatives are consistent. Because the gradient computation is done under the assumption that the overall function has a real-valued output, we treat functions with complex output in a special way. For these functions, gradcheck is applied to two real-valued functions corresponding to taking the real components of the complex outputs for the first, and taking the imaginary components of the complex outputs for the second. For more details, check out Autograd for Complex Numbers.

[9] https://pytorch.org/docs/stable/generated/torch.autograd.gradcheck.html

4. Build the Model

- We define our neural network by subclassing torch.nn.Module

```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

[3] Code Credit: https://pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html

4. Build the Model

- We define our neural network by subclassing nn.Module
- initialize the neural network layers in __init__
- Every nn.Module subclass implements the operations on input data in the forward method

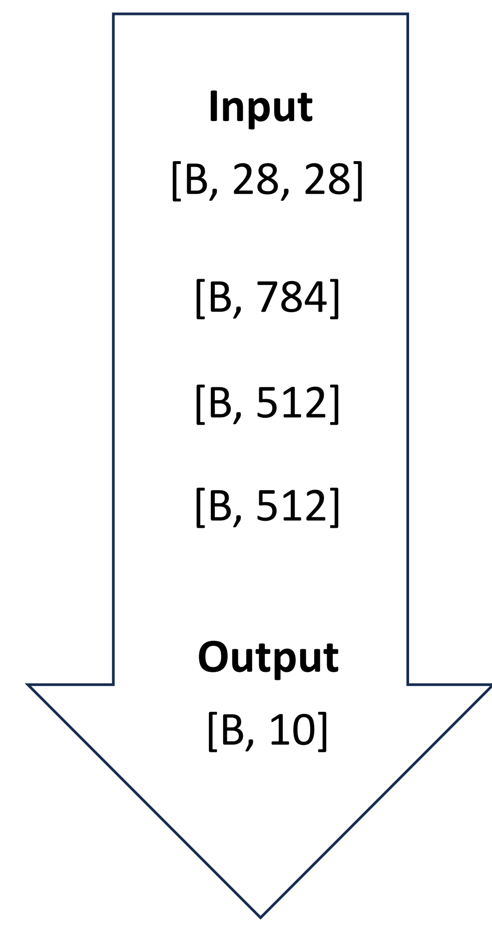
```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

[3] Code Credit: https://pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html

4. Build the Model

```
X = torch.rand(1, 28, 28, device=device)
logits = model(X)
```




```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

[3] Code Credit: https://pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html

4. Build the Model

```python
model = NeuralNetwork().to(device)
print(model)
```

```
In [12]: print(model)
NeuralNetwork(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=512, bias=True)
    (3): ReLU()
    (4): Linear(in_features=512, out_features=10, bias=True)
  )
)
```

[4] Code Credit: https://pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html

4. Build the Model

- To use the model, we pass it the input data. This executes the model's forward, along with some background operations. Do not call model.forward() directly!

```
X = torch.rand(1, 28, 28, device=device)
logits = model(X)
pred_probab = nn.Softmax(dim=1)(logits)
y_pred = pred_probab.argmax(1)
print(f"Predicted class: {y_pred}")
```

```
Predicted class: tensor([3])
```

[4] Code Credit: https://pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html

4. Case study: Linear Layer

CLASS torch.nn.Linear(*in_features*, *out_features*, *bias=True*, *device=None*, *dtype=None*) [SOURCE]

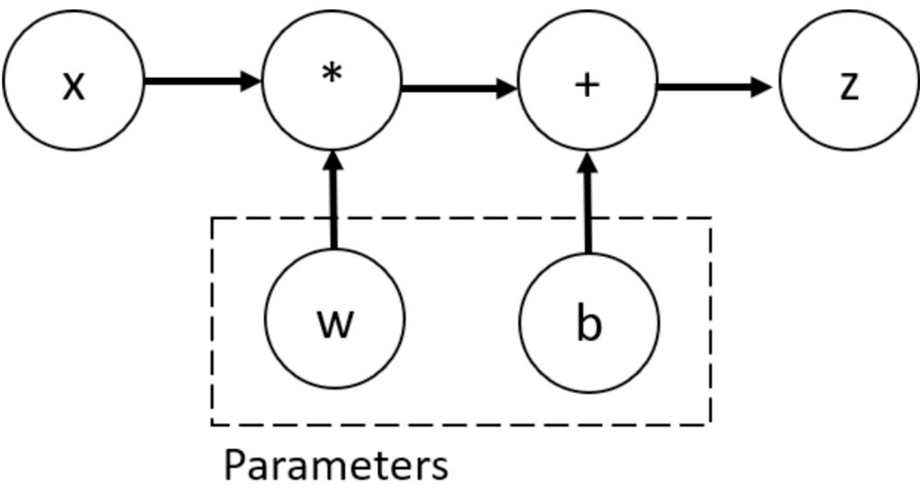
Applies a linear transformation to the incoming data: $y = xA^T + b$

This module supports TensorFloat32.

On certain ROCm devices, when using float16 inputs this module will use different precision for backward.

Parameters

- **in_features** (*int*) – size of each input sample
- **out_features** (*int*) – size of each output sample
- **bias** (*bool*) – If set to `False`, the layer will not learn an additive bias. Default: `True`



[5] Figure Credit: https://pytorch.org/tutorials/beginner/basics/autogradqs_tutorial.html

[7] Figure Credit: https://pytorch.org/docs/stable/generated/torch.nn.Linear.html

4. Case Study: Linear Layer

Zoom into nn.Linear

```
class Linear(torch.nn.Module):
```

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

4. Case Study: Linear Layer

Zoom into nn.Linear

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

```
class Linear(torch.nn.Module):
    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()
```

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

4. Case Study: Linear Layer

Zoom into nn.Linear

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

Initialization of parameters with a uniform distribution (kaiming uniform)

```
class Linear(torch.nn.Module):
    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)
```

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

4. Case Study: Linear Layer

Zoom into nn.Linear

```
class Linear(torch.nn.Module):
    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)
```

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

Initialization of parameters with a uniform distribution (kaiming uniform)

Forward call the F.linear functions

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

Tips 4.1 The backward function

Backward function is hidden in nn.Module, they are implemented in nn.functional.

These implemented standard functions is what's contributing to pytorch's autograd feature.

Let's take a look at an example of linear layer.

```python
# Inherit from Function
class LinearFunction(Function):

    # Note that forward, setup_context, and backward are @staticmethods
    @staticmethod
    def forward(input, weight, bias):
        output = input.mm(weight.t())
        if bias is not None:
            output += bias.unsqueeze(0).expand_as(output)
        return output

    @staticmethod
    # inputs is a Tuple of all of the inputs passed to forward.
    # output is the output of the forward().
    def setup_context(ctx, inputs, output):
        input, weight, bias = inputs
        ctx.save_for_backward(input, weight, bias)

    # This function has only a single output, so it gets only one gradient
    @staticmethod
    def backward(ctx, grad_output):
        # This is a pattern that is very convenient - at the top of backward
        # unpack saved_tensors and initialize all gradients w.r.t. inputs to
        # None. Thanks to the fact that additional trailing Nones are
        # ignored, the return statement is simple even when the function has
        # optional inputs.
        input, weight, bias = ctx.saved_tensors
        grad_input = grad_weight = grad_bias = None

        # These needs_input_grad checks are optional and there only to
        # improve efficiency. If you want to make your code simpler, you can
        # skip them. Returning gradients for inputs that don't require it is
        # not an error.
        if ctx.needs_input_grad[0]:
            grad_input = grad_output.mm(weight)
        if ctx.needs_input_grad[1]:
            grad_weight = grad_output.t().mm(input)
        if bias is not None and ctx.needs_input_grad[2]:
            grad_bias = grad_output.sum(0)

        return grad_input, grad_weight, grad_bias
```

[6] Code Credit: https://pytorch.org/docs/master/notes/extending.html

Tips 4.1 The backward function

Forward takes input & parameters

setup_context save input and parameters for backward

```
# Inherit from Function
class LinearFunction(Function):

    # Note that forward, setup_context, and backward are @staticmethods
    @staticmethod
    def forward(input, weight, bias):
        output = input.mm(weight.t())
        if bias is not None:
            output += bias.unsqueeze(0).expand_as(output)
        return output

    @staticmethod
    # inputs is a Tuple of all of the inputs passed to forward.
    # output is the output of the forward().
    def setup_context(ctx, inputs, output):
        input, weight, bias = inputs
        ctx.save_for_backward(input, weight, bias)

    # This function has only a single output, so it gets only one gradient
    @staticmethod
    def backward(ctx, grad_output):
        # This is a pattern that is very convenient - at the top of backward
        # unpack saved_tensors and initialize all gradients w.r.t. inputs to
        # None. Thanks to the fact that additional trailing Nones are
        # ignored, the return statement is simple even when the function has
        # optional inputs.
        input, weight, bias = ctx.saved_tensors
        grad_input = grad_weight = grad_bias = None

        # These needs_input_grad checks are optional and there only to
        # improve efficiency. If you want to make your code simpler, you can
        # skip them. Returning gradients for inputs that don't require it is
        # not an error.
        if ctx.needs_input_grad[0]:
            grad_input = grad_output.mm(weight)
        if ctx.needs_input_grad[1]:
            grad_weight = grad_output.t().mm(input)
        if bias is not None and ctx.needs_input_grad[2]:
            grad_bias = grad_output.sum(0)

        return grad_input, grad_weight, grad_bias
```

[6] Code Credit: https://pytorch.org/docs/master/notes/extending.html

Tips 4.1 The backward function

Forward takes input & parameters

setup_context save input and parameters for backward

Backward takes context and output gradient calculate the partial derivative wrt input and parameters, apply chain rule and output gradient of input and parameter

[6] Code Credit: https://pytorch.org/docs/master/notes/extending.html

```python
# Inherit from Function
class LinearFunction(Function):

    # Note that forward, setup_context, and backward are @staticmethods
    @staticmethod
    def forward(input, weight, bias):
        output = input.mm(weight.t())
        if bias is not None:
            output += bias.unsqueeze(0).expand_as(output)
        return output

    @staticmethod
    # inputs is a Tuple of all of the inputs passed to forward.
    # output is the output of the forward().
    def setup_context(ctx, inputs, output):
        input, weight, bias = inputs
        ctx.save_for_backward(input, weight, bias)

    # This function has only a single output, so it gets only one gradient
    @staticmethod
    def backward(ctx, grad_output):
        # This is a pattern that is very convenient - at the top of backward
        # unpack saved_tensors and initialize all gradients w.r.t. inputs to
        # None. Thanks to the fact that additional trailing Nones are
        # ignored, the return statement is simple even when the function has
        # optional inputs.
        input, weight, bias = ctx.saved_tensors
        grad_input = grad_weight = grad_bias = None

        # These needs_input_grad checks are optional and there only to
        # improve efficiency. If you want to make your code simpler, you can
        # skip them. Returning gradients for inputs that don't require it is
        # not an error.
        if ctx.needs_input_grad[0]:
            grad_input = grad_output.mm(weight)
        if ctx.needs_input_grad[1]:
            grad_weight = grad_output.t().mm(input)
        if bias is not None and ctx.needs_input_grad[2]:
            grad_bias = grad_output.sum(0)

        return grad_input, grad_weight, grad_bias
```

4. Build the Model: Model parameters

- Model parameters:

  Many layers inside a neural network are *parameterized*, i.e. have associated weights and biases that are optimized during training.

```
print(f"Model structure: {model}\n\n")

for name, param in model.named_parameters():
    print(f"Layer: {name} | Size: {param.size()} | Values : {param[:2]} \n")
```

```
Layer: linear_relu_stack.0.weight | Size: torch.Size([512, 784]) | Values : tensor([[-0.0137,  0.0145, -0.0072,  ...,  0.0239, -0.0182, -0.0327],
        [-0.0318, -0.0224,  0.0354,  ...,  0.0333, -0.0224,  0.0027]],
       grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.0.bias | Size: torch.Size([512]) | Values : tensor([ 0.0139, -0.0059], grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.2.weight | Size: torch.Size([512, 512]) | Values : tensor([[-0.0235,  0.0335,  0.0196,  ..., -0.0077,  0.0352, -0.0333],
        [ 0.0036,  0.0204, -0.0007,  ..., -0.0363, -0.0239, -0.0354]],
       grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.2.bias | Size: torch.Size([512]) | Values : tensor([-0.0026, -0.0323], grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.4.weight | Size: torch.Size([10, 512]) | Values : tensor([[-0.0153, -0.0410,  0.0238,  ..., -0.0146,  0.0238,  0.0026],
        [-0.0013,  0.0206, -0.0432,  ...,  0.0194,  0.0410,  0.0223]],
       grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.4.bias | Size: torch.Size([10]) | Values : tensor([-0.0231,  0.0132], grad_fn=<SliceBackward0>)
```

4. Build the Model: Model Parameters

```
Layer: linear_relu_stack.0.weight | Size: torch.Size([512, 784]) | Values : tensor([[-0.0137,  0.0145, -0.0072,  ...,  0.0239, -0.0182, -0.0327],
        [-0.0318, -0.0224,  0.0354,  ...,  0.0333, -0.0224,  0.0027]],
       grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.0.bias | Size: torch.Size([512]) | Values : tensor([ 0.0139, -0.0059], grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.2.weight | Size: torch.Size([512, 512]) | Values : tensor([[-0.0235,  0.0335,  0.0196,  ..., -0.0077,  0.0352, -0.0333],
        [ 0.0036,  0.0204, -0.0007,  ..., -0.0363, -0.0239, -0.0354]],
       grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.2.bias | Size: torch.Size([512]) | Values : tensor([-0.0026, -0.0323], grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.4.weight | Size: torch.Size([10, 512]) | Values : tensor([[-0.0153, -0.0410,  0.0238,  ..., -0.0146,  0.0238,  0.0026],
        [-0.0013,  0.0206, -0.0432,  ...,  0.0194,  0.0410,  0.0223]],
       grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.4.bias | Size: torch.Size([10]) | Values : tensor([-0.0231,  0.0132], grad_fn=<SliceBackward0>)
```

```python
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )
```