```
import torch
```

3.1 Basics of Autograd

```
# Creating a tensor with requires_grad=True
x = torch.tensor([2.0, 3.0], requires_grad=True)
print(f"Tensor x with requires_grad=True: {x}")

# Performing operations
y = x + 2
print(f"y = x + 2: {y}")

# More operations
z = y * y * 3
out = z.mean()
print(f"z = y * y * 3: {z}")
print(f"out = z.mean(): {out}")
```

```
Tensor x with requires_grad=True: tensor([2., 3.], requires_grad=True)
y = x + 2: tensor([4., 5.], grad_fn=<AddBackward0>)
z = y * y * 3: tensor([48., 75.], grad_fn=<MulBackward0>)
out = z.mean(): 61.5
```

3.2 Computing Gradients

```
# Backpropagation
out.backward()
print(f"Gradients of x: {x.grad}")

# Note: The gradient will be calculated for the scalar output 'out' w.r.t. the input 'x'
```

```
Gradients of x: tensor([12., 15.])
```

3.3 Stopping Gradient Tracking

```
# Using torch.no_grad()
with torch.no_grad():
    y = x + 2
    print(f"y = x + 2 with no_grad: {y}")

# Using detach() method
y = x.detach()
print(f"y = x.detach(): {y}")
```

```
y = x + 2 with no_grad: tensor([4., 5.])
y = x.detach(): tensor([2., 3.])
```

3.4 Gradient Accumulation

```python
# Create a tensor with requires_grad=True
x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)

# Perform first set of operations
y1 = x + 2
z1 = y1 * y1
z1 = z1.mean()

# Compute first set of gradients
z1.backward()
print(f"Gradients of x after first backward pass: {x.grad}")

# Perform second set of operations without resetting gradients
y2 = x * 3
z2 = y2.mean()

# Compute second set of gradients
z2.backward()
print(f"Gradients of x after second backward pass (accumulated): {x.grad}")

# Note: Gradients are accumulated in x.grad
# Reset gradients for a clean state
x.grad.zero_()

# Perform third set of operations
y3 = x * 4
z3 = y3.mean()

# Compute third set of gradients
z3.backward()
print(f"Gradients of x after third backward pass (after zeroing): {x.grad}")
```

```
Gradients of x after first backward pass: tensor([2.0000, 2.6667, 3.3333])
Gradients of x after second backward pass (accumulated): tensor([3.0000, 3.6667, 4.3333])
Gradients of x after third backward pass (after zeroing): tensor([1.3333, 1.3333, 1.3333])
```

3.5 Custom Gradients with Function

```python
class MyReLU(torch.autograd.Function):
    @staticmethod
    def forward(ctx, input):
        """
        The forward method computes the output of the function given the input.
        ctx is a context object that can be used to stash information for backward computation.
        input is the input tensor.
        """
        ctx.save_for_backward(input)  # Save input tensor for backward pass
        return input.clamp(min=0)  # Apply ReLU (Rectified Linear Unit) operation

    @staticmethod
    def backward(ctx, grad_output):
        """
        The backward method computes the gradient of the function w.r.t. its input.
        ctx is a context object that contains information saved during the forward pass.
        grad_output is the gradient of the loss w.r.t. the output of this function.
        """
        input, = ctx.saved_tensors  # Retrieve saved input tensor
        grad_input = grad_output.clone()  # Clone the gradient output tensor
        grad_input[input < 0] = 0  # Apply ReLU derivative: gradient is zero where input was negative
        return grad_input  # Return the gradient w.r.t. the input


# Create a tensor with requires_grad=True
x = torch.tensor([-2.0, 0.0, 3.0], requires_grad=True)

# Apply the custom ReLU function
relu = MyReLU.apply
y = relu(x)

# Perform backpropagation
y.backward(torch.tensor([1.0, 1.0, 1.0]))
print(f"Custom ReLU function gradients: {x.grad}")

```

```
Custom ReLU function gradients: tensor([0., 1., 1.])
```

Exercises

1. Implement a Custom Sigmoid Function: Create a custom autograd function for the sigmoid activation function and verify its gradients.
2. Custom Function for Squaring: Implement a custom autograd function that squares its input and compute its gradients.
3. Gradient Checking: Implement numerical gradient checking to verify the correctness of custom gradients.

```
import torch
from torch.autograd import Function

# Custom sigmoid function
class CustomSigmoid(Function):
    @staticmethod
    def forward(ctx, input):
        sigmoid_output = 1 / (1 + torch.exp(-input))
        ctx.save_for_backward(sigmoid_output)
        return sigmoid_output

    @staticmethod
    def backward(ctx, grad_output):
        sigmoid_output, = ctx.saved_tensors
        grad_input = grad_output * sigmoid_output * (1 - sigmoid_output)
        return grad_input

# Usage example
if __name__ == "__main__":
    # Input tensor
    x = torch.tensor([0.5, -1.0, 2.0], requires_grad=True)

    # Apply custom sigmoid
    custom_sigmoid = CustomSigmoid.apply
    y = custom_sigmoid(x)

    # Print the forward result
    print("Forward result:", y)

    # Compute gradients
    y.sum().backward()

    # Print the gradients
    print("Gradients:", x.grad)

```