Tensor Differentiation

Introduction

Differentiation is fundamental to the training of neural networks, as it enables optimization algorithms like gradient descent to update model parameters efficiently. When working with neural networks, the parameters (weights and biases) are often represented as tensors—multi-dimensional arrays that allow efficient computation.

Tensor differentiation refers to the process of computing derivatives of functions where inputs, outputs, and parameters are tensors. Unlike scalar differentiation, which deals with functions of single variables, tensor differentiation involves partial derivatives with respect to each element of a tensor. The result is typically another tensor of the same or different shape, often referred to as the gradient.

In neural networks, differentiation is essential for backpropagation, the algorithm used to update weights during training. Backpropagation relies on the chain rule of differentiation to compute gradients efficiently through the network's layers. The key steps involve:

1. **Forward Pass**: Compute the output of the network for a given input.
2. **Loss Computation**: Compare the predicted output with the true output using a loss function.
3. **Backward Pass (Gradient Computation)**: Compute gradients of the loss function with respect to each model parameter using automatic differentiation.
4. **Parameter Update**: Adjust weights and biases using an optimization algorithm such as stochastic gradient descent (SGD) or Adam.

The differentiation of tensor-based operations, such as matrix multiplication and activation functions, follows specific rules:

- **Element-wise operations** (e.g., ReLU, sigmoid): The gradient is computed for each element independently.
- **Matrix multiplication**: The gradient with respect to a weight matrix is determined using the chain rule and the transposition of involved matrices.
- **Softmax with Cross-Entropy Loss**: Requires special differentiation techniques to avoid numerical instability.

Frameworks like TensorFlow and PyTorch implement automatic differentiation using computational graphs, allowing efficient gradient computation without manually deriving complex derivatives.

Tensor differentiation is the foundation of neural network optimization, enabling models to learn from data by iteratively adjusting parameters based on gradients. Mastering this concept is crucial for understanding how deep learning models improve their performance over time.

Automatic Differentiation (Autograd)

Automatic differentiation (Autograd) is a key feature of modern deep learning frameworks like PyTorch. It allows for the automatic computation of gradients of functions involving tensors, enabling efficient backpropagation without requiring manual derivation of derivatives.

Autograd tracks operations performed on tensors and constructs a **computational graph**, where each node represents an operation, and edges define dependencies between operations. During the forward pass, the framework records all operations applied to tensors that require gradients. In the backward pass, it uses the **chain rule** of differentiation to compute gradients efficiently.
For a function f(x), where x is a tensor, autograd enables:

- **Forward pass**: Compute f(x).
- **Backward pass**: Compute df/dx automatically by tracing the operations.

Example in PyTorch

```
import torch

# create a tensor with requires_grad=True to track computation
x = torch.tensor(2.0, requires_grad=True)

y = x ** 3  # a function whose gradient we compute below

# compute gradient of y w.r.t. (with regard to) x
y.backward()

# print gradient dy/dx
print(x.grad)  # output: 12.0 (3 * 2^2)
```

Important Features of Autograd

- **Tracking Computation**: Autograd maintains a dynamic computation graph, allowing flexible model architectures.

- **Efficient Backpropagation**: The computational graph is used to propagate gradients in an optimized way.
- **Handling Complex Functions**: Autograd supports differentiation of scalar, vector, and tensor functions.
- **Stopping Gradient Tracking**: Operations inside `torch.no_grad()` do not track gradients, useful for inference:

```
with torch.no_grad():
    z = x * 3  # no gradients will be computed for z
```

Application in Neural Networks

In neural networks, autograd simplifies gradient computation for backpropagation. Consider a simple network trained with stochastic gradient descent (SGD):

```
import torch.nn as nn
import torch.optim as optim

# define a simple model
model = nn.Linear(1, 1)
optimizer = optim.SGD(model.parameters(), lr=0.01)

# forward pass
x = torch.tensor([[2.0]])
y_true = torch.tensor([[4.0]])
y_pred = model(x)

# compute loss
loss = (y_pred - y_true) ** 2

# backward pass (autograd computes gradients of loss w.r.t. weights and biases)
loss.backward()

# update parameters
optimizer.step()
```

Here, autograd computes gradients of the loss with respect to model parameters, which are then updated by the optimizer.

Autograd is essential for training neural networks, allowing for efficient and automated differentiation, making deep learning frameworks both powerful and user-friendly.

Gradient Approximation Using Finite Differences

In numerical optimization and scientific computing, approximating the gradient of a function is often necessary when an analytical derivative is unavailable. One of the most accurate finite difference methods for estimating gradients is the **central difference method**.
Finite difference methods estimate derivatives by using function values at nearby points. The most common finite difference approximations are:

- **Forward Difference:**

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

- **Backward Difference:**

$$f'(x) \approx \frac{f(x) - f(x-h)}{h}$$

- **Central Difference :**

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

Central Difference Approximation for Gradients

For a function $f : R^n \rightarrow R$, the gradient is given by:

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right).$$

Using central differences, the partial derivative with respect to x_i is approximated as:

$$\frac{\partial f}{\partial x_i} \approx \frac{f(x_1, \ldots, x_i + h, \ldots, x_n) - f(x_1, \ldots, x_i - h, \ldots, x_n)}{2h}.$$

The accuracy of the gradient approximation depends on the choice of h. A small h reduces truncation error but increases numerical errors due to floating-point precision limits.
For multi-dimensional functions $f(x_1, x_2, \ldots, x_n)$, the gradient can be computed using central differences for each component:

$$\nabla f \approx \left[\frac{f(x+he_1) - f(x-he_1)}{2h}, \frac{f(x+he_2) - f(x-he_2)}{2h}, \ldots, \frac{f(x+he_n) - f(x-he_n)}{2h} \right]$$

where e_i is the unit vector in the i-th direction.

Many optimization algorithms, such as gradient descent and quasi-Newton methods (e.g., BFGS), require gradient computations. When analytical gradients are unavailable, central differences provide an efficient way to approximate them with reasonable accuracy.
Despite its accuracy, central differences have limitations:

- **Computational Cost:** Requires two function evaluations per dimension, making it expensive for high-dimensional problems.
- **Numerical Precision Issues:** If *h* is too small, round-off errors can dominate.
- **Not Suitable for Noisy Functions:** In cases where function evaluations contain noise, central differences may amplify errors.

Gradient Descent with Momentum

Standard gradient descent can suffer from slow convergence and oscillations, especially in high-dimensional or highly non-convex spaces. Momentum-based methods, such as **simple momentum** and **Nesterov Accelerated Gradient (NAG)**, help accelerate convergence and stabilize the updates.
Momentum-based gradient descent helps smooth updates by incorporating past gradients into the current update step. Instead of moving solely based on the current gradient, the optimizer accumulates a velocity vector that influences future updates, similar to how a moving object gains inertia.
Let w_t represent the model parameters at iteration t, and let $\nabla L(w_t)$ be the gradient of the loss function L. The momentum update rule is:

$$v_t = \beta v_{t-1} - \eta \nabla L(w_t)$$

$$\mathrm{w}_\mathrm{t} = \mathrm{w}_\mathrm{t-1} + \mathrm{v}_\mathrm{t}$$

where:

- v_t is the velocity term, which accumulates past gradients.
- β (momentum coefficient) is a value between 0 and 1 that controls how much of the past velocity is retained.
- η (learning rate) determines the step size.

Nesterov Accelerated Gradient (NAG) is an improvement over simple momentum that anticipates the future position of the parameters before computing the gradient. Instead of applying momentum after computing the gradient, NAG first moves in the momentum direction and then computes the gradient at this **lookahead** position.

$$v_t = \beta v_{t-1} - \eta \nabla L(w_{t-1} + \beta v_{t-1})$$

$$\mathrm{w}_{\mathrm{t}} = \mathrm{w}_{\mathrm{t-1}} + \mathrm{v}_{\mathrm{t}}$$

Custom Loss Functions With Explicit Gradients

In PyTorch, a **loss function** measures how well a model’s predictions match the actual target values. While PyTorch provides built-in loss functions such as `nn.CrossEntropyLoss()` and `nn.MSELoss()`, sometimes standard loss functions are insufficient, requiring us to define **custom loss functions**.
Writing a custom loss functions allows for specifying explicit ways of computing or approximating its gradient.

In the following paragraphs, we exemplify an explicit implementation of Cross Entropy Loss. This implementation mimics PyTorch’s `nn.CrossEntropyLoss`, which internally combines `nn.LogSoftmax` and `nn.NLLLoss`.

```
class CustomCrossEntropyLoss(torch.autograd.Function):
    @staticmethod
    def forward(ctx, logits, target): # ctx = context
        # computes -log(softmax(logits)) for the correct class
        # logits = tensor of shape (batch_size, num_classes) containing features
        # target: tensor of shape (batch_size), comtaining class labels

        batch_size, num_classes = logits.shape

        # apply softmax go get predicted probablities
        softmax_probs = torch.softmax(logits, dim = 1)

        # determine the predicted probability of the correct class
        correct_class_probs = softmax_probs[torch.arange(batch_size), target]

        # compute NLL (Negative Log Likelihood)
        loss = -torch.log(correct_class_probs + 1e-9).mean() # we add a small
amount to avoid log(0)

        # save tensors for backward propagation
        ctx.save_for_backward(softmax_probs, target)

        return loss

    @staticmethod
    def backward(ctx, grad_output):
        # compute gradients of cross-entropy loss

        softmax_probs, target = ctx.saved_tensors
        batch_size = target.shape[0]

        # create one-hot encoding for target classes
        one_hot_target = torch.zeros_like(softmax_probs)
        one_hot_target[torch.arange(batch_size), target] = 1

        # compute gradient
        grad_logits = (softmax_probs - one_hot_target) / batch_size

        # scale the gradient by grad_output
        return grad_logits * grad_output, None # no gradient for target
```

This code defines a custom implementation of the cross-entropy loss using PyTorch’s `torch.autograd.Function`. It manually computes the forward and backward passes for cross-entropy loss, commonly used in classification tasks.
The class `CustomCrossEntropyLoss` extends `torch.autograd.Function` and implements the `forward` and `backward` methods.

- **Forward pass (`forward` method)**: Computes the cross-entropy loss.
- **Backward pass (`backward` method)**: Computes the gradient of the loss with regard to the logits.

Forward Pass

```
@staticmethod
def forward(ctx, logits, target):
```

- **`logits`**: A tensor of shape `(batch_size, num_classes)`, representing the raw model outputs before applying softmax.
- **`target`**: A tensor of shape `(batch_size,)` containing the class labels for each input in the batch.

```
batch_size, num_classes = logits.shape
```

This extracts the `batch_size` (number of samples) and `num_classes` (number of output classes).

```
softmax_probs = torch.softmax(logits, dim=1)
```

- Applies the **softmax function** across the second dimension (class dimension).
- Converts raw logits into **probabilities** that sum to 1 for each sample.

```
correct_class_probs = softmax_probs[torch.arange(batch_size), target]
```

- `torch.arange(batch_size)` creates a tensor of indices `[0, 1, 2, ..., batch_size - 1]`.
- Using these indices and `target`, we select the probability corresponding to the correct class for each sample.

```
loss = -torch.log(correct_class_probs + 1e-9).mean()
```

Computes the negative log likelihood (NLL), which is:

$$\text{loss} = -\frac{1}{N}\sum_{i=1}^{N}\log P(y_i \mid x_i)$$

- The small constant `1e-9` is added to avoid `log(0)`, which is undefined.
- `.mean()` ensures that the loss is averaged over the batch.

```
ctx.save_for_backward(softmax_probs, target)
```

Stores `softmax_probs` and `target` for use in the backward pass.

Backward Pass

```
@staticmethod
def backward(ctx, grad_output):
```

`grad_output`: The gradient of the loss with respect to the output (usually `1` when computing gradients during backpropagation).

```
softmax_probs, target = ctx.saved_tensors
```

Retrieves tensors saved in the forward propagation phase.

```
batch_size = target.shape[0]
one_hot_target = torch.zeros_like(softmax_probs)
one_hot_target[torch.arange(batch_size), target] = 1
```

- Creates a tensor of the same shape as `softmax_probs` filled with zeros.
- Sets the correct class positions to `1`, effectively creating a **one-hot encoding**.

Example: If `target = [1, 0, 2]` for a batch of 3 with 3 classes, the one-hot representation would be:

```
[
 [0, 1, 0],  # class 1
 [1, 0, 0],  # class 0
 [0, 0, 1]   # class 2
]
```

```
grad_logits = (softmax_probs - one_hot_target) / batch_size
```

- The derivative of the cross-entropy loss w.r.t. `logits` is:

$$\frac{\partial \text{loss}}{\partial \text{logits}} = \frac{1}{N}(\text{softmax}(x) - \text{one-hot}(y))$$

- This shows that the gradient is simply the difference between predicted probabilities and the one-hot encoded labels, scaled by `1/batch_size`.

```
return grad_logits * grad_output, None
```

- The gradient is multiplied by `grad_output` (which is usually 1 during standard backpropagation).
- The second `None` indicates that there is no gradient for `target`, as it is not a trainable parameter.

Tasks

1. **Using Autograd, compute and display the gradient of the function:**

$$f(x, y) = 3x^3 - y^2$$

2. **Consider the following function:**

$$f(x) = \sin(x)e^x$$

Compute its gradient in the following ways:

- using the analytical derivative directly:

$$\frac{df(x)}{dx} = e^x(\cos(x) + \sin(x))$$

- using backward() from Autograd
- using central differences

Measure the errors of the last two approaches compared to the first one (i.e. the Root Mean Squared Errors between the approximate gradients and the "true" gradient). Write down your observations.

3. **Minimize the function** generated with `generate_function` using autograd and gradient descent. Observe if/how increasing the complexity and dimensionality of the function influences the convergence of gradient descent.

4. **Implement gradient descent** for a fully-connected neural network using Autograd to compute gradients explicitly (i.e. `using torch.autograd.grad`).
   4.1. Modify the basic gradient descent implementation to include simple momentum. Compare the convergence to the previous (momentum-less) implementation.

5. **Modify the custom implementation of Cross Entropy Loss** so that the gradients are clipped to [-1, 1]. (i.e. the gradient values are limited to [-1, 1]). Observe and note the impact on convergence.

6. **Implement a custom MSE Loss function** with an additional scaling factor S:

   $$MSE = S\frac{1}{N}(y - \hat{y})^2$$

   Use this loss function in place of Cross Entropy Loss, with various scaling factors. Compare convergence with the previous implementation.