Core EDF code

October 5, 2024

1 Code breakdown

1.1 Forward Function

```
def Forward():
    for c in CompNodes:
        c.forward()
```

This function is responsible for performing the forward pass through the computational graph.

- `CompNodes`: This is presumably a list or collection of computed nodes (like neurons in a neural network) that are part of the model.
- `c.forward()`: This calls the forward method of each computed node `c`. In this context, the forward method calculates the output of the node based on its inputs and any associated parameters (weights, biases). The results are then passed forward to the next layer or computation in the graph.

1.2 Backward Function

```
def Backward(loss):
    for c in CompNodes + Parameters:
        c.grad = 0
    loss.grad = 1.
    for c in CompNodes[::-1]:
        c.backward()
```

Purpose: This function performs the backward pass to compute gradients for all nodes in the computational graph.

Components:

- `loss`: This argument represents the loss node, which quantifies the difference between the predicted output and the true labels.
- `c.grad = 0`: This initializes the gradient of each computed node and parameter to zero. This is important because gradients accumulate over multiple backward passes, so they must be reset for each new pass.
- `loss.grad = 1.`: This sets the gradient of the loss to 1, which is the starting point for backpropagation. It signifies that we want to compute how changes in the loss affect the parameters of the model.
- `c.backward()`: This calls the backward method for each computed node in reverse order (from output to input) to compute gradients. The gradients will be propagated backward through the network to update the parameters.

1.3 SGD Function

```
def SGD():
    for p in Parameters:
        p.value -= eta * p.grad
```

Purpose: This function updates the model parameters using the gradients computed during the backward pass, implementing the Stochastic Gradient Descent optimization algorithm.

Components:

- `Parameters`: This is presumably a collection of parameters (like weights and biases) in the model that need to be updated.
- `p.value -= eta * p.grad`: This line updates the value of each parameter `p` by subtracting the product of the learning rate `eta` and the gradient `p.grad`. This adjustment moves the parameter in the direction that minimizes the loss, based on the computed gradient.

1.4 Summary

The Forward function calculates the outputs of the network based on the current parameters. The Backward function computes gradients for all nodes in the network by propagating the loss gradient backward. The SGD function updates the model parameters using the computed gradients to minimize the loss during training. This implementation outlines a fundamental training loop in a neural network, illustrating how data flows through the network and how model parameters are updated based on the gradients obtained from the backward pass.

2 Clarification on loss.grad = 1

The line `loss.grad = 1` serves as the starting point for the backward propagation of gradients. Here's why it's done this way:

- **Loss Function Gradient:** When performing backpropagation, we want to calculate how changes in each parameter of the model affect the loss. The loss function measures the discrepancy between the predicted output and the actual output (ground truth).
- **Chain Rule in Calculus:** In the context of backpropagation, we apply the chain rule of calculus to compute gradients. The gradient of the loss with respect to itself (i.e., how the loss changes when the loss changes) is always 1. Mathematically, this can be represented as:

  $$\frac{dL}{dL} = 1$$

  where L is the loss.
- **Propagating Gradients Backward:** By setting `loss.grad = 1`, we establish a base gradient for the loss. As backpropagation progresses through the network, this base value is used to calculate how each parameter contributes to the loss. The gradients of the previous layers (i.e., the gradients of the parameters) will be computed relative to this starting point of 1.

3 Processing a Single Training Example

The three functions we provided—Forward, Backward, and SGD—can process one training example (one pair of (x, y)) at a time or can be adapted to handle a mini-batch of examples. Here's how that works:

- **Single Example:** If we call these functions with a single pair (x, y):
  - The Forward function computes the output based on that single input x.
  - The Backward function calculates the gradients based on the loss for that single output compared to the single target y.
  - The SGD function updates the parameters based on the gradients computed from that one example.

- **Batch Processing:** Typically, in practical implementations of deep learning, we often use mini-batch training, where a batch consists of multiple examples (e.g., 4, 16, 32, etc.). To adapt these functions for batch processing, the following changes would be made:
  - Forward: The computation would aggregate the outputs for all examples in the batch.
  - Backward: Gradients would be accumulated across all examples in the batch before updating the parameters.
  - SGD: The parameter updates would still be based on the average gradients computed from the batch.

3.1 Summary

`loss.grad = 1` initializes the gradient for the loss function, which simplifies backpropagation using the chain rule of calculus. The provided functions can process a single training example but can also be adjusted to handle mini-batch training by modifying how inputs and outputs are managed and how gradients are aggregated.

Let's clarify the batch processing for a batch size of 4. Within each batch:

- We first use the Forward and Backward functions for the first pair, obtaining the gradients for this one pair.
- We continue to do so for the remaining three pairs, accumulating gradients for each pair in the batch.
- After processing all four pairs, we calculate the average of gradients for each layer given the four lists of gradients.

4 Final Gradient Calculation Steps

Gradient Calculation:

For each layer (e.g., input layer, hidden layers, output layer), during the backward pass, we calculate the gradients for the parameters based on each training example. This means that, for each parameter in the layer (like weights and biases), we will accumulate the gradients from all examples in the batch.

Averaging Gradients:

After processing all examples in the batch, we compute the average gradient for each parameter in each layer:

$$\text{average gradient for parameter } p = \frac{1}{\text{batch size}} \sum_{i=1}^{\text{batch size}} \text{gradient of } p \text{ for example } i$$

This average gradient reflects the overall contribution of that parameter to the loss across all examples in the batch.

Final List of Gradients:

At the end of this process, we will have a final list of average gradients:

- **List Structure:** This list can be organized such that each entry corresponds to a parameter from the neural network, and within each entry, we have the averaged gradient for that parameter.

Let's say we have a neural network with two layers and we process a batch of 4 examples. After computing the gradients during backpropagation, we would have:

Layer 1 (Weights W_1):

- Gradients from each example: $[g_1W_1, g_2W_1, g_3W_1, g_4W_1]$
- Average gradient for W_1:

$$\text{avg } W_1 = \frac{1}{4}(g_1W_1 + g_2W_1 + g_3W_1 + g_4W_1)$$

Layer 2 (Weights W_2):

- Gradients from each example: $[g_1 W_2, g_2 W_2, g_3 W_2, g_4 W_2]$
- Average gradient for W_2:

$$\text{avg } W_2 = \frac{1}{4}(g_1 W_2 + g_2 W_2 + g_3 W_2 + g_4 W_2)$$

In this way, the model parameters are updated using the average gradients, leading to more stable and efficient training over multiple examples, making the optimization process smoother.

5 Class Input

6 Class CompNode

7 Class Parameter

8 class F(CompNode)