Course: Deep Learning **([28-11-2021])**

Submission Assignment #2

Coordinator: Jakub Tomczak *Name:* Marton Fejer, *Netid:* [2694002]

1 Question answers

Question 1 *Let f(X, Y) = X / Y for two matrices X and Y (where the division is element-wise). Derive the backward for X and for Y. Show the derivation.*

For the backward functions we use the chain rule to get the following:

$$\begin{aligned} Backward(X) &= \frac{\partial\ Loss}{\partial\ f(X, Y)} \cdot \frac{\partial\ f(X, Y)}{\partial\ X} \\ Backward(Y) &= \frac{\partial\ Loss}{\partial\ f(X,Y)} \cdot \frac{\partial\ f(X,Y)}{\partial\ Y} \end{aligned}$$

Figure 1: equation (1)

Thus, we can calculate the partial derivatives using the quotient rule to obtain the backward functions for X and Y:

$$\frac{\partial\ f(X, Y)}{\partial\ X} = \frac{\partial}{\partial X}\left(\frac{X_{ij}}{Y_{ij}}\right) = \frac{1 \cdot Y_{ij} - X_{ij} \cdot 0}{Y_{ij}^2} = \frac{Y_{ij}}{Y_{ij}^2} = \frac{1}{Y_{ij}}$$

Figure 2: equation (2)

$$\frac{\partial\ f(X, Y)}{\partial\ Y} = \frac{\partial}{\partial Y}\left(\frac{X_{ij}}{Y_{ij}}\right) = \frac{0 \cdot Y_{ij} - X_{ij} \cdot 1}{Y_{ij}^2} = \frac{-X_{ij}}{Y_{ij}^2}$$

Figure 3: equation (3)

Question 2 *Let f be a scalar-to-scalar function . Let F(X) be a tensor-to-tensor function that applies f element-wise (For a concrete example think of the sigmoid function from the lectures). Show that whatever f is, the backward of F is the element-wise application of f' applied to the elements of X, multiplied (element-wise) by the gradient of the loss with respect to the outputs.*

For the backward of f(x), we have:

$$\frac{\partial\, Loss}{\partial\, f(x)} \cdot \frac{\partial\, f(x)}{\partial\, x}$$

Figure 4: equation (4)

then, for the backward of F(x), we get the following:
Thus, the backward of F(x) would still only be an element-wise function.

$$\frac{\partial\, Loss}{\partial\, F(x_{ijk})} \cdot \frac{\partial\, F(x_{ijk})}{\partial\, x}$$

Figure 5: equation (5)

Question 3 *Let matrix W be the weights of an MLP layer with f input nodes and m output nodes, with no bias and no nonlinearity, and let X be an n-by-f batch of n inputs with f features each. Which matrix operation computes the layer outputs? Work out the backward for this operation, providing gradients for both W and X.*

For clarity, the number of features ”f” will be referred to as ”g” instead. The matrix of the weights has the shape of (f x m). The matrix describing X (so the input) has the shape of (n x g). The layer outputs must have the form of (m x g), thus in order to obtain that form, one must calculate the multiplication of the transpose of the Weights together with the Input matrix:

$$Layer\ outputs = W^T X$$

Figure 6: Matrix operation to calculate the layer outputs.

Thus, the backward for this operation wrt to W or X is the following:

$$\begin{aligned}\frac{\partial\, Loss}{\partial\, W} &= \frac{\partial\, Loss}{\partial\, Layer\ Outputs} \cdot \frac{\partial\, Layer\ Outputs}{\partial\, W}\\ \frac{\partial\, Loss}{\partial\, X} &= \frac{\partial\ Loss}{\partial\ Layer\ Outputs} \cdot \frac{\partial\ Layer\ Outputs}{\partial\ X}\end{aligned}$$

Figure 7: The backward of layer output wrt W and X.

whereby the gradient of the layer output wrt W and X is the following:

$$\begin{aligned}\frac{\partial\, Layer\ Outputs}{\partial\, W} &= \frac{\partial}{\partial W}\left(W^T X\right) = X\\ \frac{\partial\, Layer\ Outputs}{\partial\, X} &= \frac{\partial}{\partial\, X}\left(W^T X\right) = W^T\end{aligned}$$

Figure 8: Gradients of the layer output wrt W and X.

Question 4 *Let f(x) = Y be a function that takes a vector x, and returns the matrix Y consisting of 16 columns that are all equal to x. Work out the backward of f.*

The function f(x) takes the vector x of shape (n x 1) and turns it into matrix Y of shape (n x 16). Thus, the function f(x) multiplies the vector x with another vector z of shape (1 x 16):

$$f(x) = x \cdot z$$

Figure 9: Function f(x).

and thus, the backward of f(x) wrt to x is the following:

$$\begin{aligned}\frac{\partial\, Loss}{\partial\, x} &= \frac{\partial\, Loss}{\partial\, f(x)} \cdot \frac{\partial\, f(x)}{\partial\, x}\\ &= \frac{\partial\, Loss}{\partial\, f(x)} \cdot \frac{\partial}{\partial\, x}(x \cdot z) = \frac{\partial\ Loss}{\partial\ f(x)} \cdot z\end{aligned}$$

Figure 10: Backward of the function f(x) wrt x.

Question 5
1) What does c.value contain?
2) What does c.source refer to?
3) What does c.source.inputs[0].value refer to?
4) What does a.grad refer to? What is its current value?.

1) c.value contains the outcome of the tensor addition, so if a and b are tensors of the same shape, then c.value is the element-wise addition of a and b.

2) c.source refers to the operation node in the sense that it refers to what operation (together with which inputs) resulted in the node c. c.source is itself an operation node object.

3) c.source.inputs[0].value refers to the value(s) of the tensor node a. c.source.inputs[0] itself is a tensor node .

4) a.grad refers to the gradient of the loss wrt to a.value. Its current value is 0.

Question 6
You will find the implementation of TensorNode and OpNode in the file vugrad/core.py. Read the code and answer the following questions:
1) An OpNode is defined by its inputs, its outputs and the specific operation it represents (i.e. summation, multiplication). What kind of object defines this operation?
2) In the computation graph of question 5, we ultimately added one numpy array to another (albeit wrapped in a lot of other code). In which line of code is the actual addition performed?
3) When an OpNode is created, its inputs are immediately set, together with a reference to the op that is being computed. The pointer to the output node(s) is left None at first. Why is this? In which line is the OpNode connected to the output nodes?

1) An opnode performs some operation on some inputs (a tensor). The result of this operation is the output. The operation itself is a class object.

2) The addition is done with a function in core.py at line 104-108. This leads to the actual operation class "Add" which performs the actual addition at line 324.

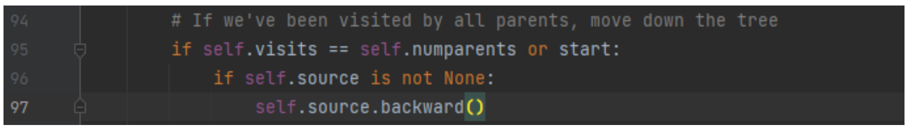
3) The reason it's left as None at first, is that it will depend on whether one does a forward or a backward pass with it (in which case they will differ). The OpNode is connected to the outputs at line 248 and 249 (first defined at 248, then connected at 249) in core.py.

Question 7
When we have a complete computation graph, resulting in a TensorNode called loss, containing a single scalar value, we start backpropagation by calling

loss.backward()

Ultimately, this leads to the backward() functions of the relevant Ops being called, which do the actual computation. In which line of the code does this happen?

This happens in line 97 (see fig. 11) whereby the "relevant operation" is self.source and the backward() function call is its backward implementation.



```
94          # If we've been visited by all parents, move down the tree
95          if self.visits == self.numparents or start:
96              if self.source is not None:
97                  self.source.backward()
```

Figure 11: Backward of the function f(x) wrt x.

.

Question 8

core.py contains the three main Ops, with some more provided in ops.py. Choose one of the ops Normalize, Expand, Select, Squeeze or Unsqueeze, and show that the implementation is correct. That is, for the given forward, derive the backward, and show that it matches what is implemented.

For normalization, the backward function has the general form:

$$Backward\big(N(x_i)\big) = \frac{\partial\, Loss}{\partial\, N(x_i)} \cdot \frac{\partial\, N(x_i)}{\partial\, x_i}$$

Figure 12: Backward of the Loss wrt x_i.

the derivative calculation is as follows:

$$\begin{aligned}\frac{\partial\, N(x_i)}{\partial\, x_i} &= \frac{\partial}{\partial\, x_i}\left(\frac{x_i}{\sum_j^n x_j}\right)\\ &= \frac{1 \cdot \sum_j^n x_j - x_i}{\left(\sum_j^n x_j\right)^2}\\ &= \frac{1}{\sum_j^n x_j} - \frac{x_i}{\left(\sum_j^n x_j\right)^2}\end{aligned}$$

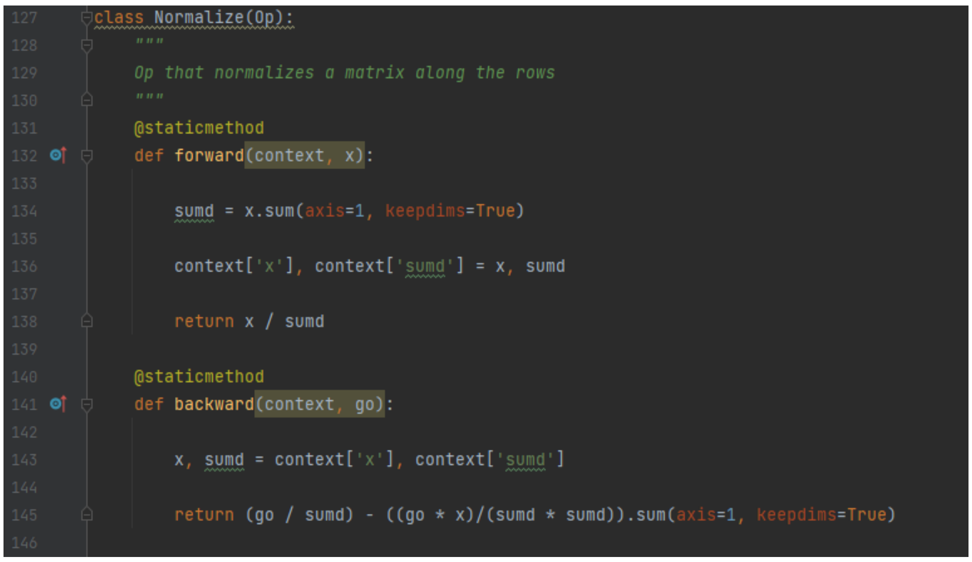
Figure 13: Derivative of the function N(x_i) wrt x_i.

Replacing the local derivative in the backward function with the calculated derivative, one gets:

$$= \frac{\frac{\partial\ Loss}{\partial\ N(x_i)}}{\sum_j^n x_j} - \frac{\frac{\partial\ Loss}{\partial\ N(x_i)} \cdot x_i}{\left(\sum_j^n x_j\right)^2}$$

Figure 14: Complete backward function of the Loss wrt x_i.

Comparing it to the backward function of normalization in ops.py, one can indeed see that it is the same:



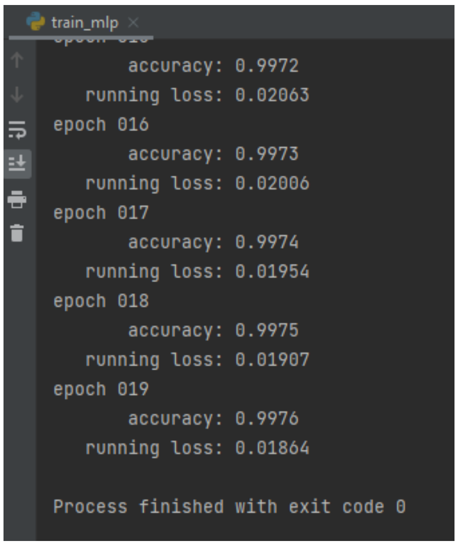
```
class Normalize(Op):
    """
    Op that normalizes a matrix along the rows
    """
    @staticmethod
    def forward(context, x):

        sumd = x.sum(axis=1, keepdims=True)

        context['x'], context['sumd'] = x, sumd

        return x / sumd

    @staticmethod
    def backward(context, go):

        x, sumd = context['x'], context['sumd']

        return (go / sumd) - ((go * x)/(sumd * sumd)).sum(axis=1, keepdims=True)

```

Figure 15: Backward function for normalization implemented in ops.py

Question 9

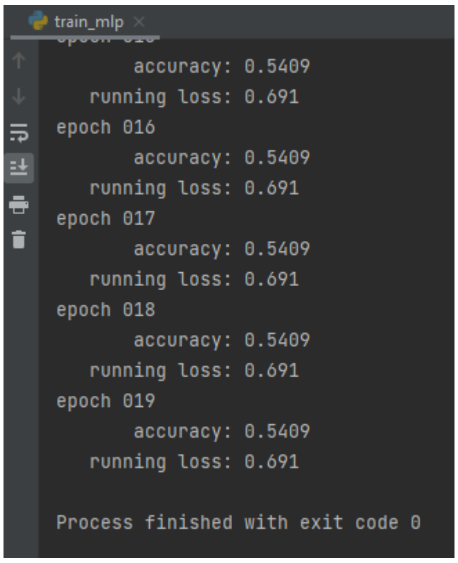
The current network uses a Sigmoid nonlinearity on the hidden layer. Create an Op for a ReLU nonlinearity (details in the last part of the lecture). Retrain the network. Compare the validation accuracy of the Sigmoid and the ReLU versions.

Retraining the network with Relu non-linearity yields a lower accuracy and higher loss for the simple synthetic dataset than the sigmoid non-linearity. In fact, it remains stagnant after the first epoch.



```
        accuracy: 0.9972
    running loss: 0.02063
epoch 016
        accuracy: 0.9973
    running loss: 0.02006
epoch 017
        accuracy: 0.9974
    running loss: 0.01954
epoch 018
        accuracy: 0.9975
    running loss: 0.01907
epoch 019
        accuracy: 0.9976
    running loss: 0.01864

Process finished with exit code 0
```

Figure 16: Training accuracy and loss with Sigmoid non-linearity.



```
        accuracy: 0.5409
    running loss: 0.691
epoch 016
        accuracy: 0.5409
    running loss: 0.691
epoch 017
        accuracy: 0.5409
    running loss: 0.691
epoch 018
        accuracy: 0.5409
    running loss: 0.691
epoch 019
        accuracy: 0.5409
    running loss: 0.691

Process finished with exit code 0
```

Figure 17: Training accuracy and loss with Relu non-linearity.