Assignment 2

Michele Vannucci (2819493)

November 2024

The code used to obtain the results of this report can be found here[1].

Answers

Part 1

question 1 Having defined the loss as l, and $f(\mathbf{X}, \mathbf{Y}) = \mathbf{X}/\mathbf{Y} = \mathbf{S}$, the derivation for $X^{\nabla}_{i,j}$ is the following:

$$
\begin{aligned}
X^{\nabla}_{i,j} &= -\frac{\partial l}{\partial X^{\nabla}_{i,j}} = \sum_{k,l} \frac{\partial l}{\partial S_{k,l}} \frac{\partial S_{k,l}}{\partial X_{i,j}} \\
&= \sum_{k,l} S^{\nabla}_{k,l} \frac{\partial \left[X_{k,l}/Y_{k,l}\right]}{\partial X_{i,j}} \\
&= S^{\nabla}_{i,j} \frac{\partial \left[X_{i,j}/Y_{i,j}\right]}{\partial X_{i,j}} \\
&= S^{\nabla}_{i,j} \frac{1}{Y_{i,j}} \quad \rightarrow \quad X^{\nabla}_{i,j} = \frac{S^{\nabla}_{i,j}}{Y_{i,j}} \quad \rightarrow \mathbf{X}^{\nabla} = S^{\nabla}/\mathbf{Y}
\end{aligned}
$$

Where the division between $\mathbf{S}^{\nabla}$ and $\mathbf{Y}$ is element-wise. Similarly for $Y^{\nabla}_{i,j}$ we get

$$
\begin{aligned}
Y^{\nabla}_{i,j} &= S^{\nabla}_{i,j} \frac{\partial \left[X_{k,l}/Y_{k,l}\right]}{\partial Y_{i,j}} \\
&= S^{\nabla}_{i,j} \cdot X_{i,j} \cdot -\frac{1}{Y^2_{i,j}} \quad \rightarrow \quad \mathbf{Y}^{\nabla} = -\left[\frac{\mathbf{S}^{\nabla} \odot \mathbf{X}}{\mathbf{Y}^2}\right]
\end{aligned}
$$

Where all the operations are performed element-wise.

question 2 Given that $F(X) = \mathbf{Y}$, the generic gradient $\mathbf{X}^{\nabla}$ is given by the following derivation:

$$
\begin{aligned}
X^{\nabla}_{ij} &= \sum_{kl} \frac{\partial l}{\partial Y_{kl}} \frac{\partial Y_{kl}}{\partial X_{ij}} \\
&= \sum_{kl} Y^{\nabla}_{kl} \frac{\partial \left[f(X_{kl})\right]}{\partial X_{ij}} \\
&= Y^{\nabla}_{ij} \cdot f'(X_{ij}) \quad \rightarrow \quad \mathbf{X}^{\nabla} = \mathbf{Y}^{\nabla} \odot F'(\mathbf{X})
\end{aligned}
$$

Where $F'(\mathbf{X})$ is a function that applies element-wise f' to the elements of $\mathbf{X}$.

[1] https://github.com/michelexyz/deeplearning_assignments

question 3 Having defined $\mathbf{Y} = \mathbf{W}\mathbf{X}^T, \quad \mathbf{W} \in \mathbb{R}^{m \times f}, \quad \mathbf{X} \in \mathbb{R}^{n \times f}$, the derivation for $\mathbf{W}^\nabla$ is the following:

$$
\begin{aligned}
W^\nabla_{i,j} &= \sum_{k,l} \frac{\partial l}{\partial Y_{k,l}} \frac{\partial Y_{k,l}}{\partial W_{i,j}} \\
&= \sum_{k,l} \frac{\partial l}{\partial Y_{k,l}} \frac{\partial \left[\sum_z W_{k,z} X^T_{z,l}\right]}{\partial W_{i,j}} \\
&= \sum_{k,l} \frac{\partial l}{\partial Y_{k,l}} \frac{\partial \left[W_{k,j} X^T_{j,l}\right]}{\partial W_{i,j}} \\
&= \sum_{l} \frac{\partial l}{\partial Y_{k,l}} \frac{\partial \left[W_{i,j} X^T_{j,l}\right]}{\partial W_{i,j}} \\
&= \sum_{l} Y^\nabla_{i,j} X^T_{j,l} = \sum_{k,l} Y^\nabla_{i,j} X_{j,l} \quad \rightarrow \quad \mathbf{W}^\nabla = \mathbf{Y}^\nabla \mathbf{X}
\end{aligned}
$$

$\mathbf{X}$ is transposed in the forward to match with the dimensions of W that is defined as $W \in \mathbb{R}^{m \times f}$, this of course affects the derivation of the gradient.

question 4 Having defined $f(\mathbf{x}) = \mathbf{Y}$ where $Y_{i,j} = x_i$ and $\mathbf{x} \in \mathbb{R}^h, \quad \mathbf{Y} \in \mathbb{R}^{h \times 16}$. The derivation for $\mathbf{X}^\nabla$ is the following.

$$
\begin{aligned}
X^\nabla_i &= \sum_j Y^\nabla_{i,j} \frac{\partial Y_{i,j}}{\partial X_i} \\
X^\nabla_i &= \sum_j Y^\nabla_{i,j} \frac{\partial X_i}{\partial X_i} \\
&= \sum_j Y^\nabla_{i,j} \cdot 1 \quad \rightarrow \quad \mathbf{X}^\nabla = \mathbf{Y}^\nabla \mathbf{1}_{16}
\end{aligned}
$$

Where $\mathbf{1}_{16}$ is a vertical vector of ones of length 16.

Part 2

question 5

1) `c.value` contains the point-wise sum of the two tensors **A** and **B**. Therefore it is an array of shape 2×2.

2) `c.source` contains a reference to the operation node (instance of the class `OpNode`) from which the matrix **C** got its value(`c.value`), in this case it represents the sum operation between **A** and **B**.

3) With `c.source.inputs[0]` we are accessing the sum operation "inputs" field. The latter contains the list of the tensor nodes used for the operation, in this case we index the first element of the list, which is the tensor object representing the matrix **A**.

4) `a.grad` represents the gradient of the loss with respect to the elements of the **A** matrix: $\mathbf{A}^\nabla$.The latter is a matrix of the same size, 2×2. Currently all its values are set to zero because we didn't run any backpropagation yet.

question 6

1) The `op` object which is an instance of the `Op` class, which is inherithed by different classes representing the low-level operations between tensors like `Add`, `Sub` and `Multiply`, is what defines the operation within the OpNode class. In detail, the `Op` class provide an interface for the **forward** and **backward** methods for the low-level operation classes as the ones mentioned above and the `do_forward` method that is discussed below.

2) The actual line of code in which the addition is performed is the last line of the forward method of the `Add(Op)` class:

```
@staticmethod
def forward(context, a, b):
    assert a.shape == b.shape, f'Arrays not the same sizes ({a.shape}
    ↪  {b.shape}).'
    return a + b
```

We arrive to this line starting from the magic function `__Add__` within the TensorNode class, where `Add`'s `do_forward`, inherited from the `Op` class, is called. This method, in turn, finally calls the `Add(Op)` forward method.

3) the `OpNode` is created within the `do_forward` function of the Op class with the line:

```
opnode = OpNode(cls, context, inputs)
```

the outputs are set right after based on the raw values given by the operation, represented by the `Add(op)` class in case of addition:

```
outputs = [TensorNode(value=output, source=opnode) for output in
↪  outputs_raw]
opnode.outputs = outputs
```

Question 7
When the backward is called on the loss tensor noode, the backword of the source `OpNode` is called. The latter then calls the given `Op` object's backward function with the line.

```
ginputs_raw = self.op.backward(self.context, *goutputs_raw)
```

The gradients are then stored in the input nodes with the lines:

```
for node, grad in zip(self.inputs, ginputs_raw):

    assert node.grad.shape == grad.shape, f'node shape is {node.size()} but
    ↪  grad shape is {grad.shape}'

    node.grad += grad
```

Finally, the process is repeated recursively, making sure that all the parents nodes have been visited before calling the backward on a child node.

Question 8 I bring the example of applying the `squeeze` function on a 2d matrix $\mathbf{X}$ for which the second axis has length 1 ($\mathbf{X} \in \mathbb{R}^{h \times 1}$) so that $S(\mathbf{X}) = \mathbf{Y}, \mathbf{Y} \in \mathbb{R}^{h}$, where S applies the `squeeze` forward operation. The latter can be defined as applying element-wise the function $s(X_{i,j}) = Y_i$ with $Y_i = X_{i,j}$. Since the values of the input are left unchanged , only

the formalization of the size is changed, the derivation is straightforward:

$$X^{\nabla}_{i,j} = Y^{\nabla}_{i} \cdot \frac{\partial Y_i}{\partial X_{i,j}}$$
$$X^{\nabla}_{i,j} = Y^{\nabla}_{i} \cdot \frac{\partial X_{i,j}}{\partial X_{i,j}}$$
$$X^{\nabla}_{i,j} = Y^{\nabla}_{i}$$
$$.$$

This could be derived analogously, squeezing any other dimension, that has to be saved in the context during the forward. This means that the gradient of the input is equal to the gradient of the output only, we only have to add the dimension that was removed during the backward, which size will still be 1. This is exactly what it's done in *vugrad*'s implementation:

```
class Squeeze(Op):
    """
    Remove a specific singleton dimension
    """

    @staticmethod
    def forward(context, input, dim=0):
        context['dim'] = dim

        return input.squeeze(dim)

    @staticmethod
    def backward(context, goutput):
        dim = context['dim']

        return np.expand_dims(goutput, axis=dim)
```

question 9 I implemented the ReLU operation in the following way:

```
class ReLU(Op):

    @staticmethod
    def forward(context, input):

        context['mask'] = input > 0

        return np.maximum(0, input)

    @staticmethod
    def backward(context, goutput):

        return goutput * context['mask']
```

With this I implemented the same MLP network by changing the line where the sigmoid was applied, within the forward function of the module, with:

```
hidden = ReLU.do_forward(hidden)
```

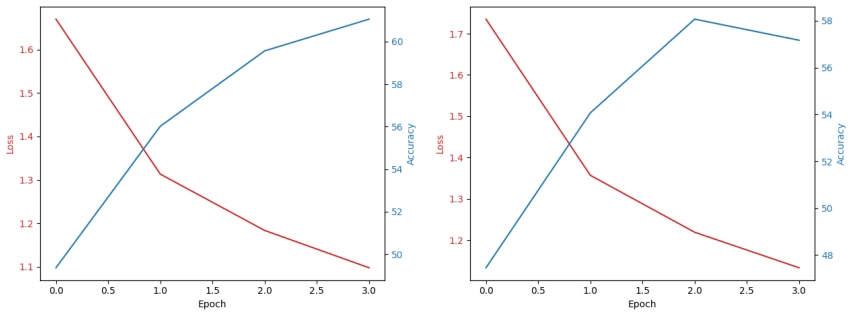
The full implementation can be looked at in the Appendix. Comparing the two networks, Table 1 shows the results training the network over the mnist data and 10 epochs.

Table 1: Results training the network over the mnist data and 10 epochs. Different learning rates and activation functions have been used.

Activation	Learning rate	Training running loss	Validation accuracy
ReLU	$0,0001$	$0,1583$	$95,64\%$
Sigmoid	$0,0001$	0.1023	$96,46\%$
ReLU	$0,001$	2.301	$10,58\%$
Sigmoid	$0,001$	$0,2904$	$94,12\%$

It's interesting to notice how the ReLU function struggles a lot with a learning rate of $0,001$ this can be motivated by the fact that the gradient's magnitude is preserved with this non-linearity and this learning rate is to high for the network to learn properly.

question 11 After implementing the classifier I tried see if using Nesterov stochastic gradient descent would improve the performance. Figure 1 shows the results with and without over four epochs. A momentum of $0,09$ and a learning rate of $0,001$ has been used .



(a) Performances without using Nesterov (b) Performances using Nesterov

Figure 1: Training loss and validation accuracy with and without Nesterov stochastic gradient descent over four epochs. A momentum of $0,09$ and a learning rate of $0,001$ has been used .

As an example the code to define the optimizer for the Nesterov implementation is the following:

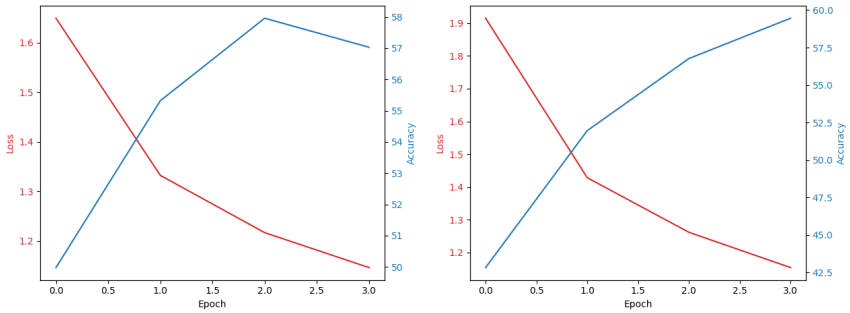
```
net = Net()

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9,
↪  nesterov=True)

train_losses, test_accuracies = train_model(net, trainloader, testloader,
↪  criterion, optimizer, epochs=4)
```

question 12 For this question I tried tweaking the weight decay parameter, to introduce regularizaton and possibly improve the performance on the validation set. I also experimented adding the batch normalization at different levels of the network, and changing the batch size. Finally, I also tried increasing the number of epochs even though it was obviously increasing the computation time dramatically since I was running on CPU this is why I didn't go above 8 epochs. Figure 2 shows some results for different batch sizes.

The graph in figure 2b suggests that a higher batch size could improve the performance with more epochs.



(a) Performances using Nesterov and a batch size of 2

(b) Performances using Nesterov and a batch size of 10

Figure 2: Training loss and validation accuracy with Nesterov stochastic gradient descent over four epochs. A momentum of $0,09$ and a learning rate of $0,001$ has been used and batch size of 2 and 10 respectively.

Adding a 2d and 1d batch normalization before applying the non linearties didn't improve the performance significantly with the initial settings of Figure 1. Finally, the best validation accuracy, 63.51% was reached with 8 epochs and using a weight decay of 0.001 while the rest of the settings where left the same as the implementation used in Fig. 1.

Appendix

Question 9

The full implementation of the network with the ReLU activation function is the following:

```
class MLPReLU(vg.Module):
    """
    A simple MLP with one hidden layer, and a sigmoid non-linearity on the
    ↪  hidden layer and a softmax on the
    output.
    """

    def __init__(self, input_size, output_size, hidden_mult=4):
        """
        :param input_size:
        :param output_size:
        :param hidden_mult: Multiplier that indicates how many times bigger
        ↪  the hidden layer is than the input layer.
        """
        super().__init__()

        hidden_size = hidden_mult * input_size
        # -- There is no common wisdom on how big the hidden size should
        ↪  be, apart from the idea
        #    that it should be strictly _bigger_ than the input if at all
        ↪  possible.

        self.layer1 = vg.Linear(input_size, hidden_size)
        self.layer2 = vg.Linear(hidden_size, output_size)
        # -- The linear layer (without activation) is implemented in
        ↪  vugrad. We simply instantiate these modules, and
        #    add them to our network.

    def forward(self, input):

        assert len(input.size()) == 2

        # first layer
        hidden = self.layer1(input)

        # non-linearity
        hidden = ReLU.do_forward(hidden)
        # second layer
        output = self.layer2(hidden)

        # softmax activation
        output = vg.logsoftmax(output)
        # -- the logsoftmax computes the _logarithm_ of the probabilities
        ↪  produced by softmax. This makes the computation
        #    of the CE loss more stable when the probabilities get close to
        ↪  0 (remember that the CE loss is the logarithm
        #    of these probabilities). It needs to be implemented in a
        ↪  specific way. See the source for details.
```

```
        return output

    def parameters(self):

        return self.layer1.parameters() + self.layer2.parameters()
```