Assignment 2

Dexter Chen (2774239)

November 2023

1 Answers

question 1 Given $Z_{i,j} = \frac{X_{i,j}}{Y_{i,j}}$:

$$\frac{\partial f}{\partial X_{i,j}} = \frac{\partial Z_{i,j}}{\partial X_{i,j}} = \frac{1}{Y_{i,j}}$$

$$\frac{\partial f}{\partial Y_{i,j}} = \frac{\partial Z_{i,j}}{\partial Y_{i,j}} = -\frac{X_{i,j}}{(Y_{i,j})^2}$$

question 2 Given the loss as $\boldsymbol{L}$ and the output tensor of $Y_{i,j} = f(X_{i,j})$, the backward pass of $\boldsymbol{F}$ involves computing the gradient of a scalar loss with respect to the input tensor $\boldsymbol{X}$:

$$\frac{\partial L}{\partial X_{i,j}} = \frac{\partial L}{\partial Y_{i,j}} \cdot \frac{\partial Y_{i,j}}{\partial X_{i,j}}$$

Since $Y_{i,j} = f(X_{i,j})$, we have: $\frac{\partial Y_{i,j}}{\partial X_{i,j}} = f'(X_{i,j})$ By the chain rule:

$$\frac{\partial L}{\partial X_{i,j}} = \frac{\partial L}{\partial Y_{i,j}} \cdot f'(X_{i,j})$$

question 3 The layer output $\boldsymbol{Y}$ is computed by multiplying the input matrix $\boldsymbol{X}$ with the weight matrix $\boldsymbol{W}$.

The gradient of the loss $\boldsymbol{L}$ with respect to $\boldsymbol{W}$ is given by:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial Y} \cdot \frac{\partial Y}{\partial W}$$

Since $\boldsymbol{Y} = \boldsymbol{X} \cdot \boldsymbol{W}$, the partial derivative $\frac{\partial Y}{\partial W}$ is the transpose of $\boldsymbol{X}$: X^T

$$\frac{\partial L}{\partial W} = X^T \cdot \frac{\partial L}{\partial Y}$$

The gradient of the loss $\boldsymbol{L}$ with respect to $\boldsymbol{X}$ is given by:

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y} \cdot \frac{\partial Y}{\partial X}$$

Similarly, the partial derivative $\frac{\partial Y}{\partial X}$ is the transpose of $\boldsymbol{W}$: W^T

$$\frac{\partial L}{\partial Y} = \frac{\partial L}{\partial Y} \cdot W^T$$

question 4 Given that $f(x) = Y$ and $Y_{i,j} = x_i$, where Y is a matrix with dimension of $n \times 16$, compute $\frac{\partial L}{\partial x}$:

$$\frac{\partial L}{\partial x} = \sum_{j=1}^{16} \frac{\partial L}{\partial Y_{i,j}}$$

The gradient with respect to x_i is the sum of the gradients of the loss with respect to each element (total of 16) in the column i of Y.

question 5 Given the size of $(2, 2)$, create TensorNode ***a,b,c***.

1) ***c.value*** contains an array of size $(2, 2)$, with each element equals to the sum of ***a*** and ***b***.

2) ***c.source*** refers to the OpNode(s) that created the TensorNode ***c***

3) ***c.source.inputs[0].value*** refers to the first element of $c.source$. Here as ***A+B=C*** , it represent the raw value of A.

4) ***a.grad*** stores the gradient of ***loss*** wrt to ***a***, which is initially set to all zero with the same dimensions as the value.

question 6 Based on the **core.py** file:

1) The *operation* is defined by the ***op*** class with two static methods, **forward** and **backward**.

2) At **line 169**, the code:

```
node.grad += grad
```

performances the addition process for gradients.

3) Because the output nodes are not yet computed at the time of **OpNode** creation. It is connected to the output nodes in the **do_forward** method of the **Op** Class at **line 249**.

question 7 The actual computation of the backward pass happens in the **backward** method of the **OpNode** class at **line 159**:

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

Here, **self.op** is the operation that needs to be performed, **self.context** is the context that was saved during the forward pass, and **goutputs_raw** are the gradients of the former outputs.

question 8 Chosen the **Normalize** operation for this task, the forward pass is **y = x / sumd**, where **sumd** is the sum of each row of the input matrix **x**. The derivative of **y** with respect to the direct contributor **x** is **1 / sumd** and **-x / sumd*sumd** wrt. **sumd**. In the **backward** operation,

```
def backward(context, go):
    x, sumd = context['x'], context['sumd']
    return (go / sumd) - ((go * x)/(sumd * sumd)).sum(axis=1, keepdims=True)
```

go / sumd correspond to the derivative part of **1/sumd**. The **((go * x)/(sumd * sumd)).sum(axis=1, keepdims=True)** is same to the derivatives of **-x / sumd*sumd**. The combination of these two parts correctly computes the gradients of the **Normalize** operation wrt. the input matrix **x**.

question 9 Similarly, add the ReLU **Op** to the **ops.py** as:

```
class ReLU(Op):
    """
    Op for element-wise application of ReLU function
    """
    @staticmethod
    def forward(context, input):
        relu = np.maximum(0,input)# -- store the ReLU of x for the backward pass
        context['relu'] = relu
        return relu
    @staticmethod
    def backward(context, goutput):
        relu = context['relu']
        return goutput * (relu > 0)# -- ReLU gradient is 1 if input is positive, 0 otherwis
```

Change the method in **train_mlp_ReLU.py** and retrain the network on the synthetic data. After 20 epochs, the validation accuracy of the Sigmoid is 0.9894 and that of the ReLU is 0.9915, showing an accuracy improvement.

question 10 With one more hidden linear layer

```
    ...
    self.layer1 = vg.Linear(input_size, hidden_size,init_method)
    self.layer2 = vg.Linear(hidden_size, hidden_size,init_method)
    self.layer3 = vg.Linear(hidden_size, output_size,init_method)
    ...
```

and the initialize weights set to a standard normal distribution

```
...
def __init__(self, input_size, output_size, init_method):
    super().__init__()
    if init_method == 'zeros':
        w = np.zeros((output_size, input_size))
        self.w = TensorNode(w)
    elif init_method == 'normal':
        w = np.random.randn(output_size, input_size)
        self.w = TensorNode(w)
    else:
        glorot_std = np.sqrt(2.0 / (input_size + output_size))
        w = np.random.randn(output_size, input_size) * glorot_std
        self.w = TensorNode(w)
...
```

the validation accuracy of the synthetic data has a mere increase to 0.9951 and with initialize weights to zeros, the corresponding accuracy after 20 epochs drops to 0.5476. The improvement of accuracy in standard normalized initial weights is also observed in MINST data. However, the conflicting results from zeros initial weights are still questionable.

question 11 The built classifier helps to identify the subject in picture (CIFAR10). Here, **tune** from **Ray** library is applied to 'play around' the hyperparameters of **lr, batch_size, epoch**. Generally, the deepening of the network layers have a more positive effect; increasing the number of cycles of training has a greater positive effect on the training of the network model within a certain range; a suitable and smaller batch size can improve the training effect with a certain amount of data; a varying learning rate may play a better role, but it is also difficult to set. For example, when **lr** set to 0.001, the accuracy increases with more epochs (**epoch**i100), whereas with higher learning rates (0.005, 0.01, 0.015), accuracy decreases after the peak. The over all tuned parameters are as **lr=0.005, epoch=60** with accuracy of 0.7601

question 12 Here, Residual Network(ResNet) is applied to build the classifier. Residual Block, defined in the **ResidualBlock** class, is a fundamental building block in ResNet. It consists of two convolutional layers and in the classifier there are 8 such blocks, building 18 fully connected layers(ResNet18). The output of these layers is added to the original input, forming a "shortcut" or "residual" connection.

```
...
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1, downsample=None):
        super(ResidualBlock, self).__init__()
        self.conv1 = conv3x3(in_channels, out_channels, stride)
        self.bn1 = nn.BatchNorm2d(out_channels)
```

```
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(out_channels, out_channels)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.downsample = downsample

    def forward(self, x):
        residual = x
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.bn2(out)
        if self.downsample:
            residual = self.downsample(x)
        out += residual
        out = self.relu(out)
        return out
...
```

With the step of 1000, after 3 Epochs, the loss is 0.5736; after 10 epochs, the loss is 0.3724 and continually deceases with more epochs until converge. Eventually, after 50 epochs, the loss decrease to 0.1014 and the overall accuracy on the test image set is 0.892 , which is much more accurate than previous models.

2 Appendix

All source code can be accessed on my [GitHub]. Due to system limitation, GPUs are not accessible. All computations are performed on CPU, thus the codes are modified to comply with such limitation. Some results are not printed out but can be accessed easy by running the code.