CS 577 — Deep Learning — Final exam practice problems

1 Automatic differentiation for scalars

Please read these instructions:

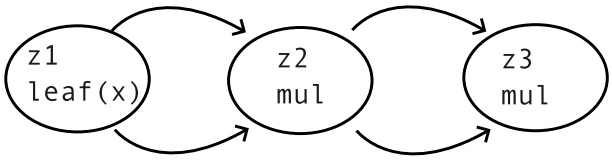
- Orient your graph such that inputs are (roughly) on the left and outputs are on the right, including intermediate nodes.
- For nodes that have multiple inputs such as `mul`, place the inputs (roughly) from top to bottom, where top is the "left" input and bottom is the "right" input.
- You will be provided the **non-reversed** topological ordering of the nodes. It will be labeled as "`label_name:op_name`".

1.1 The quartic: x^4

Draw the computational graph for the following function. Then compute `x.grad` using backpropagation.

```
x = ag.Scalar(2.0, label="z1:leaf(x)")
z1 = x
z2 = z1*z1
z3 = z2*z2
z3.backward()
print(x.grad)
```

Solution to problem 1.1:
The **computational graph** and the **topological order** are as follow:



`topo_order: [z1:leaf(x), z2:mul, z3:mul]`

The **forward computation** are as follow: `z1.value = 2` (by the definition of `x`), `z2.value = 4` and `z3.value = 16`.
Backpropagation computation:
Iteration 0 (of backward): The backward function during the 0-th iteration of backward is

```
        def __mul__(self, other):
            output = ag.Scalar(self.value * other.value, inputs=[self, other])
            def _backward():
                self.grad += other.value * output.grad
                other.grad += self.value * output.grad
```

where

- `output` is `z3`
- `self` and `other` are both `z2`

Hence, we have

```
z2.grad += z2.value * z3.grad
z2.grad += z2.value * z3.grad
```

and so

```
z2.grad == 8
```

after the 0-th iteration of backward.

Iteration 1: The operation is `mul` again, however

- `output` is `z2`
- `self` and `other` are both `z1`

Hence, we have

```
z1.grad += z1.value * z2.grad
z1.grad += z1.value * z2.grad
```

and so

```
z1.grad == 32
```

after the 1-st iteration of backward.

Iteration 2: There is no operation at `z1`, which is a leaf node (`inputs = []` is the empty list). The backward pass is finished.

Thus, the final answer `x.grad` is equal to `32`.

(Note: it is acceptable to simply say that "During Iteration 2, the output node is a leaf node where there is no op. Skipping.")

1.2 $\log(\exp(xy))$

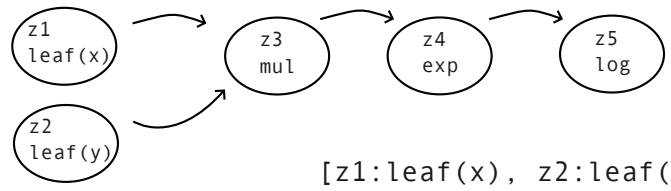
Draw the computational graph for the following function. Then compute `x.grad` and `y.grad` using backpropagation.

```
x = ag.Scalar(2.0, label="z1:leaf(x)")
y = ag.Scalar(3.0, label="z2:leaf(y)")

z1 = x
z2 = y
z3 = z1*z2
z4 = ag.exp(z3)
z5 = ag.log(z4)
z5.backward()
print(x.grad, y.grad)
```

Solution to problem 1.2:

The computational graph and the (non-reversed) topological ordering is given below:



`[z1:leaf(x), z2:leaf(y), z3:mul, z4:exp, z5:log]`

Forward computation: `z1.value = 2` and `z2.value = 3` (by the definition of `x` and `y`), `z3.value = 6`, `z4.value = exp(6)` and `z5.value = log(exp(6)) = 6`.

Iteration 0: The operation is `log`. The output node is `z5` and the input node is `z4`. The backpropagation calculation is

```
z4.grad += (1/z4.value) * z5.grad
```

Note that when backward is called on `z5`, we set `z5.grad` to the value `1`. Thus

```
z4.grad += (1/exp(6)) * 1
```

And so the value of `z4.grad` is `1/exp(6)` at the end of iteration 0.

Iteration 1: The operation is `exp`. The output node is `z4` and the input node is `z3`. The backpropagation calculation is

```
z3.grad += exp(z3.value) * z4.grad
```

and so

```
z3.grad += exp(6) * (1/exp(6))
```

And so the value of `z3.grad` is `1` at the end of iteration 1.

Iteration 2: The operation is `mul`. The output node is `z3` and the input nodes is `z1` and `z2`. The backpropagation calculation is

```
z1.grad += z2.value * z3.grad
z2.grad += z1.value * z3.grad
```

and so after iteration 2 `z1.grad == z2.value == 3` and `z2.grad == z1.value == 2`.

In **iteration 3 and 4**, the output nodes are leaf nodes so there are no ops.

Final answer, `z1.grad == x.grad == z2.value == 3` and `z2.grad == y.grad == z1.value == 2`.

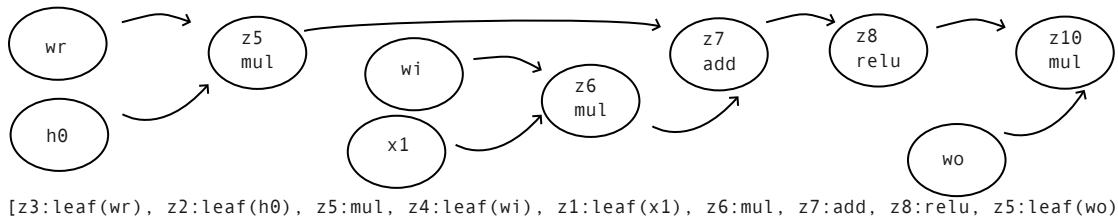
1.3 Recurrent neural networks

Draw the computational graph for the following function. Then compute `wr.grad`, `wi.grad`, and `wo.grad` using backpropagation.

```
x1 = ag.Scalar(2.0, label="z1:leaf(x1)")
h0 = ag.Scalar(3.0, label="z2:leaf(h0)")
wr = ag.Scalar(4.0, label="z3:leaf(wr)")
wi = ag.Scalar(5.0, label="z4:leaf(wi)")
wo = ag.Scalar(6.0, label="z5:leaf(wo)")

z1 = x1
z2 = h0
z3 = wr
z4 = wi
z5 = z3*z2 # wr*h0
z6 = z4*z1 # wi*x1
z7 = z5+z6
z8 = ag.relu(z7) # relu(wr*h0 + wi*x1)
z9 = wo
z10 = z8*z9
z10.backward()

print(wr.grad, wi.grad, wo.grad)
```




```
[z3:leaf(wr), z2:leaf(h0), z5:mul, z4:leaf(wi), z1:leaf(x1), z6:mul, z7:add, z8:relu, z5:leaf(wo), z10:mul]
```

The **forward computation**:

```
z5 = 12 # wr*h0
z6 = 10 # wi*x1
z7 = 22 # wr*h0 + wi*x1
z8 = 22 # relu(wr*h0 + wi*x1)
z10 = 132 # wo*relu(wr*h0 + wi*x1)
```

1.4 Recurrent Neural Networks: Solution

Topological Order:

$z3$: leaf(wr), $z2$: leaf(h0), $z5$: mul, $z4$: leaf(wi), $z1$: leaf(x1), $z6$: mul, $z7$: add, $z8$: relu, $z9$: leaf(wo), $z10$: mul

Iteration 0: Output node is `z10`. The op is `mul` and so

```
z8.grad += z9.value * z10.grad
z9.grad += z8.value * z10.grad
```

Substituting in values:

```
z8.grad = 6*1 = 6
z9.grad = 22*1 = 22
```

Iteration 1: Output node $z9$ is a leaf, skipped.
Iteration 2: Output node $z8$: relu. The op is `relu`.
Since `z7.value > 0`, the derivative is:

```
z7.grad += z8.grad
```

and so

```
z7.grad = 1*6 = 6
```

Iteration 3: Output node $z7$. The op is `add`:

```
z5.grad += z7.grad
z6.grad += z7.grad
```

and so

```
z5.grad = 6
z6.grad = 6
```

Iteration 4: Output node $z6$. The op is `mul`.

```
z4.grad += z1.value*z6.grad
z1.grad += z4.value*z6.grad
```

Substitute in

```
z4.grad = 2*6 = 12
z1.grad = 5*6 = 30
```

Iteration 5 and 6: Output nodes `z1` and `z4` are leaves. Skipped.
Iteration 7: Output node $z5$. Op `mul`. Thus

```
z3.grad += z2.value * z5.grad
z2.grad += z3.value * z5.grad
```

Substituting in values:

```
z3.grad = 3*6 = 18
z2.grad = 4*6 = 24
```

Iteration 8 and 9: Output nodes `z2` and `z3` are leaves. Skipped.
Final Answer: The computed gradients are:

```
wr.grad = z3.grad = 18
wi.grad = z4.grad = 12
wo.grad = z9.grad = 22
```

2 Convolutional networks

2.1 `im2col`

What is the shape of the `im2col_mat_sparse` matrix constructed below? Give your answer in terms of `N`, `Cin`, `Hin`, `Win`, `K` and `S`.

```
def im2col_matrix_sparse(Xin, K, S=1):
    N, Cin, Hin, Win = Xin.shape
    CHW = Cin * Hin * Win
    Hout = (Hin - K)//S + 1
    Wout = (Win - K)//S + 1
    P = Hout * Wout  # Total number of patches per image
    patch_size = Cin * K * K # Size of each flattened patch

    data = [1 for _ in range(P*patch_size)]
    row_indices = []
    col_indices = list(range(P*patch_size))

    patch_idx = 0
    for hout in range(Hout):
        for wout in range(Wout):
            for cin in range(Cin):
                for hker in range(K):
                    for wker in range(K):
                        input_index = cin * Hin * Win + hout * S * Win + wout * S + hker *
    Win + wker
                        row_indices.append(input_index)
            patch_idx += 1

    im2col_mat_sparse = csr_matrix((data, (row_indices, col_indices)), shape=(CHW, P *
    patch_size))
    return im2col_mat_sparse
```

Solution:
The shape is

```
(Cin * Hin * Win, Hout * Wout * Cin * K * K)
```

where we let

```
Hout = (Hin - K)//S + 1
Wout = (Win - K)//S + 1
```

2.2 Convolution layer time and space complexities

- What is the time complexity of computing `Xout_flat` in the line

```
Xout_flat = (Xin_patches_flat @ self.weight) + self.bias
```

from the code block below? Give your answer in terms of `N`, `Cin`, `Cout`, `Hin`, `Win`, `K` and `S`.

Note: this question is specifically about the computational complexity of the line of code that produces `Xout_flat`, excluding the lines of code leading up to it.

- How much memory is required to store `Xin_im2col.value` (note the change from previous version to be specifically about the values and not the grads)? Give your answer in terms of bytes. (Assume entries of the array are in float64, which requires 8 bytes)

```
class Conv2d(Module):
    def __init__(self, in_channels, out_channels, kernel_size, stride=1):
        super().__init__()
        self.in_channels = in_channels # Cin
        self.out_channels = out_channels # Cout
        self.kernel_size = kernel_size # K
        self.stride = stride # S

        kaiming_he_init_constant = np.sqrt(2 / (in_channels * kernel_size**2))

        self.weight = ag.Tensor(np.random.randn(in_channels*kernel_size**2, out_channels
) * kaiming_he_init_constant)

        self.bias = ag.Tensor(np.zeros(out_channels))

        self._parameters['weight'] = self.weight
        self._parameters['bias'] = self.bias

        self.im2col_mat = None # im2col_mat will be cached

    def forward(self, Xin): # Xin.shape == (N, Cin, Hin, Win)
        N, Cin, Hin, Win = Xin.shape
        assert(Cin == self.in_channels)

        K = self.kernel_size
        S = self.stride
        Hout = (Hin - K) // S + 1
        Wout = (Win - K) // S + 1
        P = Hout * Wout  # Total number of patches per image
        patch_size = Cin * K * K  # Size of each flattened patch
        Cout = self.out_channels

        Xin_flat = Xin.reshape(-1, Cin * Hin * Win)

        if self.im2col_mat is None: # Cache the im2col matrix
            self.im2col_mat = im2col_matrix_sparse(Xin, K, S)

        Xin_im2col = ag.spcmatmul(Xin_flat, self.im2col_mat)
        Xin_patches_flat = Xin_im2col.reshape(N, P, patch_size)

        Xout_flat = (Xin_patches_flat @ self.weight) + self.bias
        Xout_flat = ag.moveaxis(Xout_flat, 1, 2)
        Xout = Xout_flat.reshape(N, Cout, Hout, Wout)

        return Xout # Xout.shape == (N, Cout, Hout, Wout)
```

Solution (time complexity):
`Xin_patches_flat` has shape

```
(N, P, Cin * K * K)
```

where we let

```
P = Hout * Wout
Hout = (Hin - K)//S + 1
Wout = (Win - K)//S + 1
```

`self.weight` has shape

```
(Cin * K * K, Cout)
```

The computational complexity of this line is dominated by the (tensorized) matrix multiplication between two tensors of shape

```
(N, P, Cin * K * K)      and      (Cin * K * K, Cout)
```

Thus, the computational complexity is

```
O(N * P * Cin * K * K * Cout)
```

Solution (space complexity): `Xin_im2col.value` has shape

```
(N, P * Cin * K * K)
```

where

```
P = Hout * Wout
Hout = (Hin - K)//S + 1
Wout = (Win - K)//S + 1
```

Thus, the memory required to store the value tensor is

```
8*N* P * Cin * K * K    (bytes)
```

3 Attention

- How much memory is required to store the following intermediate activations (i.e., the value tensors excluding the grad tensors) in `SingleHeadAttention` portion of the transformer block?: `Queries, Keys, KQ, expKQ, softmaxKQ, P`. (note the change from previous version to be specifically about the values and not the grads)
- How about the `MLP`? Namely how much memory is required to store the following?: `hidden`

```
class SingleHeadAttention:
    def __init__(self, n_features):
        self.Wq = ag.Tensor(np.random.randn(n_features, n_features), label="Wq")
        self.Wk = ag.Tensor(np.random.randn(n_features, n_features), label="Wk")
        self.Wv = ag.Tensor(np.random.randn(n_features, n_features), label="Wv")
    def __call__(self, Xin):
        # Xin is a (n_samples, n_context, n_features) tensor
        # Xout is *also* a (n_samples, n_context, n_features) tensor
        Queries = Xin @ self.Wq
        Keys = Xin @ self.Wk
        KQ = (Keys @ ag.moveaxis(Queries, 1,2))
        expKQ = ag.exp(KQ)
        softmaxKQ = expKQ / ag.sum(expKQ, axis=1, keepdims=True)
        P = ag.moveaxis(Xin,1,2) @ softmaxKQ
        Xout = ag.moveaxis(P, 1,2) @ self.Wv
        return Xout

class MLP:
    def __init__(self, n_features, n_hidden):
        self.Wh = ag.Tensor(np.random.randn(n_features, n_hidden), label="Whidden")
        self.bh = ag.Tensor(np.random.randn(n_hidden), label="bhidden")
        self.wo = ag.Tensor(np.random.randn(n_hidden, n_features), label="Wout")
        self.bo = ag.Tensor(np.random.randn(n_features), label="bout")

    def __call__(self, Xin):
        hidden = ag.relu((Xin @ self.Wh) + self.bh)
        return hidden @ self.wo + self.bo

class TransformerBlock:
    def __init__(self, n_features, n_hidden):
        self.att = SingleHeadAttention(n_features)
        self.mlp = MLP(n_features, n_hidden)
    def __call__(self, Xin):
        return self.mlp(self.att(Xin))
```

Solution (the self attention portion):

```
Queries.shape = Keys.shape = (n_samples, n_context, n_features)
Memory required = n_samples * n_context * n_features * 8
```

```
KQ.shape = expKQ.shape = P.shape = softmaxKQ = (n_samples, n_context, n_context)
Memory = n_samples * n_context^2 * 8
```

Solution (the MLP portion):

```
hidden.shape = (n_samples, n_context, n_hidden)
Memory = n_samples * n_context * n_hidden * 8
```

4 Tensor rematerialization (aka checkpointing)

4.1 No rematerialization

How much memory in bytes (removed the requirement that the answer be in megabytes) does computing the `forward` function below require for the `.value` fields for the intermediate activations (so excluding the input and the output `ag.sum(x)`)? Leave your answer in algebraic form, i.e., without substituting in actual numbers for `num_layers`, `num_samples`, and `dim_hidden`.

```
    class Tensor: # Tensor with grads
        def __init__(self,
                     value,
                     requires_grad=False,
                     rematerializer = None, # None means don't rematerialize, i.e., keep
                     op="",
                     _backward= lambda : None,
                     inputs=[],
                     label=""):

            if type(value) in [float ,int]:
                value = np.array(value)

            self.requires_grad = requires_grad
            self.rematerializer = rematerializer

            self.value = 1.0*value
            self.grad = None

            if self.requires_grad:
                self.grad = np.zeros_like(self.value)

# [...]

num_layers = 10
num_samples = 4096
dim_hidden = 1000

weights = [ag.Tensor(0.02*np.random.randn(dim_hidden, dim_hidden),
                     requires_grad = True) for _ in range(num_layers)]
X = ag.Tensor(np.random.randn(num_samples, dim_hidden))

def forward(x, weights):
    for w in weights:
        x = ag.matmul(x, w)
    return ag.sum(x)
```

Solution:
Each intermediate activation

```
x.value.shape = (num_samples, dim_hidden)
```

Memory required for activation per layer

```
num_samples * dim_hidden * 8
```

With

```
num_layers = 10
```

the **final answer for** memory required for activation across all layers is

```
num_layers * num_samples * dim_hidden * 8
```

4.2 With rematerialization

One way to implement rematerialization for the model considered in the previous part is to pass a "rematerializer" to the intermediate tensor to reconstruct the `.value` field:

```
def forward_with_rematerializer(x, weights, checkpoints):

    farthest_checkpoint = 0
    x_at_farthest_checkpoint = x

    for i, w in enumerate(weights):
        if i in checkpoints:
            x = ag.matmul(x, w)
            farthest_checkpoint = i
            x_at_farthest_checkpoint = x
        else:
            def _rematerializer():
                xval = x_at_farthest_checkpoint.value
                for w in weights[farthest_checkpoint:(i+1)]:
                    xval = np.matmul(xval,w.value)
                return xval
            x = ag.matmul(x, w)
            x.rematerializer = _rematerializer

    return ag.sum(x)
```

Once the `.value` field is no longer needed during the forward computation, it is discarded:

```
# inside ag.Tensor class definition:
        def discard_value_if_has_rematerializer(self):
            if self.rematerializer is not None:
                self.value = None
            return None
```

How much memory (in bytes, left in algebraic form as above) is used for allocating the intermediate `x.value` fields if we call `forward_with_rematerializer` with the following inputs at the end of the forward pass?

```
num_layers = 10
num_samples = 4096
dim_hidden = 1000

weights = [ag.Tensor(0.02*np.random.randn(dim_hidden, dim_hidden),
                     requires_grad = True) for _ in range(num_layers)]
x = ag.Tensor(np.random.randn(num_samples, dim_hidden))
checkpoints = [5]
```

Solution:

As before, the memory required for activation per layer is

```
num_samples * dim_hidden * 8
```

However, for a layer that is not a checkpoint, the activation is discarded immediately afterwards, thus the memory usage is.

```
len(checkpoints) * num_samples * dim_hidden * 8
```

where `len(checkpoints) = 1` in our case.