CS 577 — Deep Learning — Final exam practice problems

1 Automatic differentiation for scalars

1.1 x^4

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

```python
x = ag.Scalar(2.0, label="z1\nleaf(x)")
z1 = x
z2 = z1*z1
z3 = z2*z2
z3.backward()
print(x.grad)
```

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

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

```python
x = ag.Scalar(2.0, label="z1\nleaf(x)")
y = ag.Scalar(3.0, label="z2\nleaf(y)")

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

1.3 Recurrent neural networks

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

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

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`.

```python
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
```

2.2 Convolution layer time and space complexities

- What is the time complexity of computing `Xout_flat` in the 2D convolution layer defined below? Give your answer in terms of `N, Cin, Cout, Hin, Win, K` and `S`.
- How much memory is required to store `Xin_im2col`? Give your answer in terms of bytes. (Assume entries of the array are in float64, which requires 8 bytes)

```python
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)
```

3 Attention

- How much memory is required to store the following intermediate activations in `SingleHeadAttention` portion of the transformer block?: `Queries`, `Keys`, `KQ`, `expKQ`, `softmaxKQ`, `P`.
- How about the `MLP`? Namely how much memory is required to store the following?: `hidden`

```python
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))
```

4 Tensor rematerialization (aka checkpointing)

4.1 No rematerialization

How much memory (in MB) 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)`)?

```python
    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)
```

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 MB) is used for allocating the `.value` fields if we call `forward_with_rematerializer` with the following inputs?

```
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]
```