Lewis Arnsten

Problem 1:

Layer	Tensor dimensions	Parameter count	FLOPs
Input	32 x 32 x 3	0	0
Conv F=16, 5 x 5	32 x 32 x 16	(5 * 5 * 3 + 1) * 16	2457600
ReLU	32 x 32 x 16	0	16348
Conv F=24, 3 x 3	32 x 32 x 24	(3 * 3 * 16 + 1) * 24	7077888
ReLU	32 x 32 x 24	0	24576
Pool	16 x 16 x 24	0	6144
Conv F=32, 3 x 3	16 x 16 x 32	(3 * 3 * 24 + 1) * 32	3538944
ReLU	16 x 16 x 32	0	8192
Conv F=48, 3 x 3	16 x 16 x 48	(3 * 3 * 32 + 1) * 48	7077888
ReLU	16 x 16 x 48	0	12288
Pool	8 x 8 x 48	0	3072
Conv F=64, 3 x 3	8 x 8 x 64	(3 * 3 * 48 + 1) * 64	3538944
ReLU	8 x 8 x 64	0	4096
Conv F=64, 1 x 1	8 x 8 x 64	(1 * 1 * 64 + 1) * 64	524288
ReLU	8 x 8 x 64	0	4096
Pool	4 x 4 x 64	0	1024
Conv F=96, 3 x 3	4 x 4 x 96	(3 * 3 * 64 + 1) * 96	1769472
ReLU	4 x 4 x 96	0	1536
Pool	2 x 2 x 96	0	384
Conv F=128, 3 x 3	2 x 2 x 128	(3 * 3 * 96 + 1) * 128	884736
ReLU	2 x 2 x 128	0	512
Pool	1 x 1 x 128	0	128

a) In our convolutional layers, since we are adding an amount of zero-padding that allows for placement of the filter center over each spatial location ((F-1)/2), size is preserved spatially. Thus, the width and height of the activation tensor produced by each convolutional layer will be equal to the width and height of the input to that layer. The number of output channels of each convolutional layer will be equal to the number of filters (F). A convolutional layer's parameter count is calculated via: (kernel_size^2 * input_channels + 1) * filters. Activation functions and pooling layers have no parameters. For convolutional layers, the number of FLOPs is calculated via: (2 * kernel_size^2 * input_channels * output_channels * H * W). For max pooling layers, the number of FLOPs is calculated via: (output_H * output_W * C). For ReLU activation functions, the only operation performed is a max() function for each output neuron. Thus, the number of FLOPs is just calculated via: (H * W * C).

b) Conv F=64, 3 x 3 is the shallowest layer for which each spatial location in its output tensor has a full receptive field with respect to the network input.

c) We can replace each pooling layer in our network with a convolutional network with a stride of two. For these convolutional layers, we will continue to add zero-padding that allows for placement of the filter center over each spatial location. Here are the changes that must be made to our network:

Original pooling layer	New convolutional layer
Pool (16 x 16 x 24)	Conv F=24, kernel = 2 x 2, stride = 2
Pool (8 x 8 x 48)	Conv F=48, kernel = 2 x 2, stride = 2
Pool (4 x 4 x 64)	Conv F=64, kernel = 2 x 2, stride = 2
Pool (2 x 2 x 96)	Conv F=96, kernel = 2 x 2, stride = 2
Pool (1 x 1 x 128)	Conv F=128, kernel = 2 x 2, stride = 2

Problem 2:

The ideal activation function to place after batch normalization in this deep fully-connected network is ReLU. As none of the listed options are ReLU activations, I will separate the listed options based on whether or not they would be acceptable choices. For, some of the listed choices would cause the network to not learn at all.

<u>The following activation functions would be acceptable choices:</u>

4. act(x) = exp(x) / (exp(x)+1) → This is equivalent to Sigmoid activation. It is an acceptable activation function. This activation function is better than all the other listed options for this network (it acts similarly to ReLU when x > 0).

1. act(x) = min(0,x) → This activation function is the opposite of ReLU. It will cause all values greater than 1 to be set to 0. This is a valid activation function, though it would likely be more effective if it was -min(0,x).
5. act(x) = sign(x) → This activation function is similar to binary step, except its output is -1 when x < 0. It is an acceptable activation function. Though it should be noted that it is not an ideal choice as its derivative cannot be calculated at 0.

<u>The following activation function would not be acceptable choices:</u>

2. act(x) = $1/\sqrt{2} * x$→ This is a linear function and therefore useless as an activation function.
3. act(x) = ReLU(x - 10) → Since activation happens after batch normalization, subtracting 10 from inputs will make all inputs negative. Then, the result of ReLU activation (max(0,x)) will be 0 for all inputs.

Problem 3:

a) What goes wrong with the strategy of initializing all newly added parameters to 0?

Since we are using ReLU activation, when the input to our activation function is 0 the output will also be zero. In other words, max(0,x) = 0 when x = 0. Subsequently, all the local gradients (dL / dW) for the weights will be 0. Thus, the network activations will collapse to zero and the network will not learn.

To further demonstrate why this network will not learn, consider a forward pass for this n-layer network. If all weights are initialized to 0, the output of every layer will be zero (due to zero-valued weights and subsequent ReLU activation). Thus, the inputs to every layer except the first will also be 0.

b) Design a correct initialization for the parameters of the expanded model.

As discussed in lecture, we will use Kaiming / MSRA initialization for the parameters of the expanded model. Thus, we will initialize all newly added parameters to be random numbers. We will then apply a ReLU correction to all newly added parameters (multiply by sqrt(2 / input_dimension)).

ReLU correction: std = sqrt(2 / input_dimension)
New weights = random_initialization * std

Problem 4:

a) input $x: T \times D$ trainable parameters $\rightarrow \gamma, \beta$

$$\mu_j = \frac{1}{T}\sum_{t=1}^{T} x_{i,j}$$

$$\sigma_j^2 = \frac{1}{T}\sum_{t=1}^{T}(x_{i,j} - \mu_j)^2$$

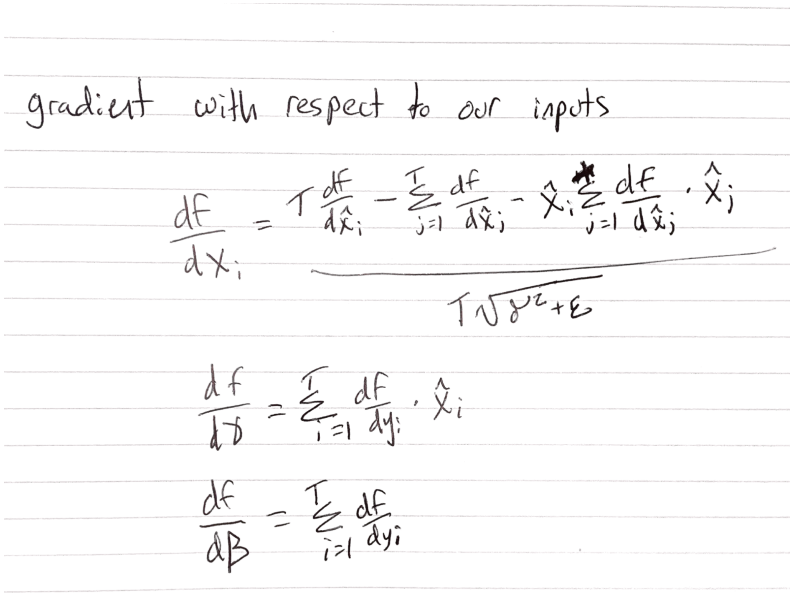
$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

$$y_{i,j} = \gamma \hat{x}_{i,j} + \beta_j$$

b) Here I will provide forward and backward computation passes for this modified sequence normalization layer that utilize running estimates of mean and variance. At each timestep I will update my running averages for mean and variance via a momentum parameter (I based this on batch normalization during testing, which requires running mean and variance).

```
def sequencenorm_forward(x, gamma, beta, parameters):
    e = parameters[“epsilon”]
    momentum = parameters[“momentum”]

    T, D = x.shape
    mu = mean(x, axis=0)
    variance = variance(x, axis=0)

    if exist parameters['running_mean'] and parameters[‘running_variance’]:
        running_mean = parameters['running_mean']
        running_variance = parameters[‘running_variance’]
        running_mean = momentum * running_mean + (1 - momentum) * mu
        mu = running_mean
        running_variance = momentum * running_var + (1 - momentum) * variance
        variance = running_variance
    else:
        running_mean = zeros(T, dtype=x.dtype)
        running_variance = zeros(T, dtype=x.dtype)

    x_mu = x - mu
    #squared = x_mu * x_mu
    #variance = 1./T * sum(squared, axis = 0)
    sqrt_var = sqrt(variance + e)
    inv_var = 1./sqrtvar
    xhat = x_mu * inv_var
```

```
    out = gamma * xhat + beta

    cache = (xhat, gamma, inv_var)

    parameters['running_mean'] = running_mean
    parameters['running_variance'] = running_variance

  return out, cache
```




```
def sequencenorm_backward(out, cache):
    T, D = out.shape

    xhat, gamma, inv_var = cache
    dx_hat = out * gamma

    dx = 1.0/T * inv_var * (T*dx_hat - sum(dx_hat, axis=0) - xhat*sum(dx_hat*xhat, axis=0))
    dbeta = sum(out, axis=0)
    dgamma = sum(xhat*out, axis=0)

    return dx, dgamma, dbeta
```

c) Sequence normalization cannot be used in transformer architecture. Transformer blocks include layer normalization, not batch normalization. In our definition of sequence normalization, we considered elements of a single sequence as comprising a batch. We cannot consider elements of a single sequence as comprising a layer.

Problem 5:

a) Our goal is to train our RNN on long input sequences (length T), however we have limited memory (memory budget of $O(\log_2(T))$) for storing hidden state activations. We will compensate for this limited memory by re-computation. Essentially, we will sacrifice computational efficiency to reduce memory consumption. Instead of storing all hidden states, hidden states will be restored when they are needed by executing an extra forward operation.

Outline: When errors have to be backpropagated from time t to (t - 1), the RNN state can be re-evaluated by executing a forward operation that takes the previous hidden state as input. The backward operation can follow. Consider our final sequence of states $h_1,h_2,...,h_t$. To make my implementation as efficient as possible, I will do a series of forward passes to find the midpoint of my sequence. I will then recurse to the left and right until I reach the base case, storing the states at the midpoints as I do.

```
def recursive(forward, h0, begin, end):
    if T - begin == 1:
        return h0
    else:
        h = h0
        diff = (end-begin)//2
        for i in range(diff):
            h = forward(h)
        for s in recursive(forward, h, begin+diff, end):
            yield h
        for s in recursive(forward, h0, begin, b+diff):
            yield h


Wh = initialize random weights
Wx = initialize random weights
h0 = initialize all zeros
h = tanh(dot(Wh, h0) + dot(Wx, x))
h0 = h
for i in range(T):
    x, h = forward(x, h)
h_last = h
grad = all ones in shape of h
for h in recursive(forward, h0, 0, T):
    grad = grad * backward(h)
```

b) I will now consider the backpropagation of errors on a simple RNN which uses a fixed amount of memory. As we have a constant memory budget M, all states that cannot be stored (T > M) must be restored when they are needed. For example, a state at time t will be re-evaluated by

forward-propagating inputs starting from the beginning until time t. Thus, we will repeat t forward steps before backpropagating gradients one step backwards. Thus, this algorithm will require t *(t + 1) / 2 forward passes to backpropagate over t time steps. For a large T, BPTT is therefore $O(n^2)$ in time.

Problem 6:

a) We are replacing a convolutional layer with a module that runs RNNs horizontally and vertically across the spatial dimensions of an input tensor. X is an input tensor with spatial extent H x W and C channels. We have two RNNs, RNN A which runs across the rows of X producing H^A, and RNN B, which runs across the columns of H^A producing H^B.

```
def RNN_forward(x, h_prev, Wx, Wh):
  h_next = tanh(dot(Wh, h_prev) + dot(Wx, x))
  #cache to save values for backward pass
  cache = (x, h_next, h_prev, Wh,  Wx)
  return h_next, cache

X = input tensor
h0 = initialize random
Wx = initialize random
Wh = initialize random

cacheA = {}
cacheB = {}
H, W, C = x.shape
Initialize H^A and H^B to all zeros of shape (H, W, C)

for i in range(H):
  if i == 0:
        H^A[i,:,:], cacheA[str(i)]  = RNN_forward(x[i, :, :], h0, Wx, Wh)
  else:
        H^A[i,:,:], cacheA[str(i)]  = RNN_forward(x[i, :, :],  H^A[i-1,:, :], Wx, Wh)

for i in range(W):
  if i == 0:
        H^B[:,i,:], cacheB[str(i)]  = RNN_forward(H^A[:, i, :],  h0, Wx, Wh)
  else:
        H^B[:,i,:], cacheB[str(i)]  = RNN_forward(H^A[:, i, :],   H^B[:,i-1, :], Wx, Wh)

y_pred = softmax(dot(V,H^B) + bias)
```

b) We are given upstream gradient tensor G of shape H x W x C.

```
RNN_backward(g_next, cache):
        (x, h_next, h_prev, Wh, Wx) = cache
        #gradient of tanh
        h_deriv = (1 - h_next * h_next) * g_next
        dh_prev = dot(h_deriv, Wh.T)
        dWh = dot(h_prev.T, h_deriv)
        dx = dot(h_deriv, Wx.T)
        dWx = dot(x.T, h_deriv)
        return dx, dh_prev, dWx, dWh

H, W, C = G.shape
Initialize dx, dWx, and dWh with zeros, add extra dimension at index 0
Initialize dh_prev with zeros

for i in reversed(range(W)):
   dh_curr = dh_prev + G[:,i,:]
   dx[i], dh_prev, dWx[i], dWh[i] = RNN_backward(dh_curr, cacheB[str(i)])

for i in reversed(range(H)):
   dh_curr = dh_prev + G[i,:,:]
   dx[i], dh_prev, dWx[i], dWh[i] = RNN_backward(dh_curr, cacheA[str(i)])

dWx = sum(dWx, axis=0)
dWh = sum(dWh, axis=0)
```

c) At spatial location (i,j) in the output tensor H^B, RNN A has run across i rows of the input tensor and RNN B has run across j columns of the input tensor. Thus, the spatial location (i,j) has a receptive field of size i * j.

d) The number of floating point operations in a forward pass of the proposed module will depend on the spatial dimensions of the input tensor. Our module consists of two RNNs, each of which contains an addition operation, two dot products (between vectors and matrices), and an activation function. A dot product between a vector of size n and a matrix of size (m x n) performs 2*m*n FLOPs. Compared to the FLOPs of the dot products, the FLOPs of the addition operation and activation function are insignificant. Thus, I estimate the FLOPs in a forward pass of our RNN to be H*2*(2*H*W) + W*2*(2*H*W).

The total number of FLOPs of a convolution layer with filters of 3 x 3 spatial extent and the same number of input and output channels can be estimated via: 3*3*3*H*W*3.

Thus, the number of FLOPs in our module is significantly greater than the number of FLOPs in the proposed convolutional layer.

e) A convolutional layer can capture spatial features from an image. It can learn from the arrangement of pixels and relationship between them in the image. The proposed module does not have the ability to capture spatial features from the image. However, the proposed module will consider every image row as a sequence of pixels. It will then consider the columns of the image as a sequence of rows. These recurrent connections are our modules key advantage over a classic convolutional network.