Here are some practice problems to work through in preparation for Exam 2.

**Problem 1 (0 points)** In formatting training data for our models to perform a task like machine translation, we took inputs $x_{1:N} \in \mathcal{V}^*$ and target outputs $y_{1:M} \in \mathcal{V}^*$, and formatted them as follows:

$$[\text{START_X}] \;\; x_1, x_2, \ldots, x_N \;\; [\text{START_Y}] \;\; y_1, y_2, \ldots, y_M \;\; [\text{END}] \tag{1}$$

That whole sequence we can denote as a sequence $w_{1:(N+M+3)}$. Passing the whole sequence to a model we're training, we get per-token losses $-\log p_\theta(w_i \mid w_{<i})$ for each $i = 2, \ldots, (N+M+3)$. (We don't get the first likelihood, of the `[START]` token, as it conditions on nothing and our architectures produce a distribution over the next word for each prefix given as input.) You can think of this vector of log-likelihoods as

$$p = \left[-\log p_\theta(w_2 \mid w_1); -\log p_\theta(w_3 \mid w_{<3}); \cdots; -\log p_\theta(w_{M+N+3} \mid w_{<(M+N+3)})\right] \tag{2}$$

To compute our goal likelihood terms, we often multiply $p$ with a **mask** consisting of zeros and ones, in order to sum only the terms we want to optimize over.

First, write the mask $z$ that computes the term $-\log p_\theta(y_{1:M} \mid x_{1:N})$ when elementwise multipled as $z \odot p$ (that is, $z_1 p_1, z_2 p_2, \ldots$)

Second, write the mask $z$ that computes the term $-\log p_\theta(x_{1:N}, y_{1:M})$.

**Solution:**

For $p(y \mid x)$, we have the following:

$$p = [\underbrace{0, \ldots, 0}_{\text{N}}, \underbrace{0}_{1}, \underbrace{1, \ldots, 1}_{\text{M}}, \underbrace{1}_{1}, ] \tag{3}$$

where the first $N$ zeros mask out the terms corresponding to $p(x_i \mid x_{<i})$, the next 0 masks the likelihood of `[START_Y]` given $x$, and then the next $M$ ones are for the terms $-\log p(y_i \mid y_{<i}, x)$, and the last 1 is for the likelihood of the `END` token.
Note that this comes from

$$-\log p_\theta(y_{1:M} \mid x_{1:N}) = \sum_{j=1}^{M} -\log p(y_j \mid y_{<j}, x_{1:N} \tag{4}$$

For $p(x, y)$, we have

$$p = [\underbrace{1, \ldots, 1}_{\text{N}}, \underbrace{1}_{1}, \underbrace{1, \ldots, 1}_{\text{M}}, \underbrace{1}_{1}, ] \tag{5}$$

**Problem 2 (0 points)** Consider an elementwise multiply of two matrices $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{m \times n}$. Assume that $A$ and $B$ are represented with 32-bit floating-point numbers. Compute (1) the number of memory accesses, (2) the number of floating-point operations, and (3) the arithemtic intensity of the elementwise multiply of $A$ and $B$:

$$A \cdot B \tag{6}$$

(Recall that $(A \cdot B)_{ij} = A_{ij}B_{ij}$.)

**Solution:**

Here are our memory accesses:
Read $A$ : $mn$ reads, each of 4 bytes, so $4mn$.
Read $B$ : $mn$ reads, each of 4 bytes, so $4mn$.
(Write result) : $mn$ writes, each of 4 bytes, so $4mn$. In lecture we didn't include writes, so it's up to you whether you want to.
Here are our floating point operations: Compute $A_{ij}B_{ij}$ for all $i$,$j$. This is $mn$ floating-point operations.
So, if we include writes, that's an arithmetic intensity of:

$$\frac{mn}{4mn + 4mn + 4mn} = \frac{1}{12} \tag{7}$$

**Problem 3 (0 points)** Repeat Problem 2, but with 16-bit floating-point operations, and 8-bit floating-point operations.

**Solution:**

We have the same number of floats to read, but in 16 bits, each only reads $2mn$ bytes. The number of floating-point operations is the same. So we have

$$\frac{mn}{2mn + 2mn + 2mn} = \frac{1}{6} \tag{8}$$

That's better! And for 8 bits, it's $\frac{1}{3}$.

**Problem 4 (0 points)** Given the answer to Problem 3, and our desire for high arithemtic intensity, why don't we always use, say, 8, or 4, or 2, or 1-bit floating-point representations?

**Solution:**

With lower precision, the error with which we compute our gradients and internal computation grows; there just aren't enough values to express the kinds of distinctions we want (like computing $\frac{1}{n}$ an average over a long sequence length), so it's a tradeoff.

**Problem 5 (0 points)** Recall the language model from Lecture 1:

$$h_{t-1}^{\text{avg}} = \frac{1}{t-1} \sum_{j=1}^{t-1} E_{w_j} \tag{9}$$

$$p_\theta(w_t \mid w_{<t}) = \text{softmax}(E h_{t-1}^{\text{avg}})_{w_t} \tag{10}$$

Consider that I have the sequence

$$h_1^{(0)}, \ldots, h_{t-1}^{(0)} = E_{w_1}, \ldots, E_{w_{t-1}} \tag{11}$$

That's my first layer of my neural network. The second layer of the neural network is equal to:

$$h_1^{(1)}, \ldots, h_{t-1}^{(1)} = \text{SelfAttention}(h_1^{(0)}, \ldots, h_{t-1}^{(0)}) \tag{12}$$

Define matrices $W_Q, W_K, W_V \in \mathbb{R}^{d \times d}$ for single-headed self-attention such that for all $j = 1, \ldots, t-1$, we have that:

$$h_j^{(1)} = h_j^{\text{avg}} \tag{13}$$

That is, write down the matrices such that self-attention using those matrices implements a simple average of the prefix so far.

**Solution:**

We have that $W_Q = 0$, (that is, the matrix of all zeros), likewise $W_K = 0$, and $W_V = I$, the identity matrix.
(We'd expect you to show why this works out.)

**Problem 6 (0 points)** In this question we'll work through why we want nicely-normed inputs to self-attention for the sake of learning.

I have a sequence of vectors which I'm about to apply self-attention to:

$$h_1^{(0)}, h_2^{(0)}, h_3^{(0)} \tag{14}$$

For simplicity, let the self-attention matrices be $W_Q = W_K = W_V = I$, all the identity matrix. Applying self-attention, we get:

$$h_1^{(1)}, h_2^{(1)}, h_3^{(1)} = \text{SelfAttention}(h_1^{(0)}, h_2^{(0)}, h_3^{(0)}) \tag{15}$$

Ok, now let's assume the following about our input vectors:

$$h_1^{(0)} = 1000 h_3^{(0)} \tag{16}$$
$$h_2^{(0)} = 0.1 h_3^{(0)} \tag{17}$$
$$\tag{18}$$

First, **compute the attention distribution** for word 3, that is, compute $\alpha_{3,j}$ for $j = 1, 2, 3$.

Second, uh oh, it turns out that I really wanted to have larger $\alpha_{3,2}$, that is, to have word 3 look more at word 2. Describe why it's difficult to do this.

**Solution:**

The distribution is $[1, 0, 0]$, that is, $\alpha_{3,1} = 1$, $\alpha_{3,2} = 0$, $\alpha_{3,3} = 0$. this is because the softmax inputs are $[1000x, 0.1x, x]$ for some $x$. As long as $x$ isn't tiny (maybe we should have stipulated that $\|h_3\| = 1$,) , this leads to the resulting distribution. It's hard to make $\alpha_{3,2}$ larger because, as we showed in homework one, the gradient of the softmax vanishes when you have probability near 0 on an element. Since the gradient is approximately zero, it will be hard (or impossible if the gradient is exactly 0) to learn via gradient descent to make $\alpha_{3,2}$ larger.