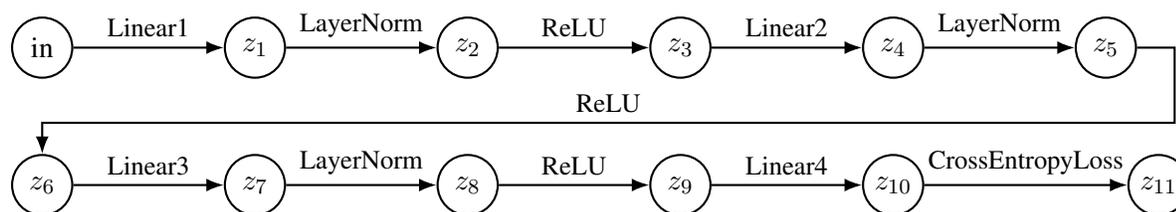


This discussion will cover normalization and regularization.

## 1. Model Size

Consider the following classification model architecture:



Suppose we know some of the dimensions of the model parameters as follows:

- `in.shape = [batchsize=4, in_feats=52]`
- `Linear1(in_feats=52, out_feats=_____)`
- `Linear2(in_feats=128, out_feats=256)`
- `Linear3(in_feats=_____, out_feats=20)`
- `Linear4(in_feats=_____, out_feats=_____)`
- `num_classes=1000`

- (a) Fill in the missing dimensions above. Additionally, label the diagram above with the dimensions of the intermediate activations ( $z_1$  through  $z_{10}$ ) and output ( $z_{11}$ ). Assume that `CrossEntropyLoss` calculates average loss across the entire batch.

### Solution:

- `Linear1(in_feats=52, out_feats=128)`
- `Linear3(in_feats=256, out_feats=20)`
- `Linear4(in_feats=20, out_feats=1000)`
- `z1, z2, z3 shape = [4, 128]`
- `z4, z5, z6 shape = [4, 256]`
- `z7, z8, z9 shape = [4, 20]`
- `z10 shape = [4, 1000]`
- `z11 shape = [] (scalar)`

Note that the shape of the output of `CrossEntropyLoss` can vary depending on library implementation and arguments passed in. By default, **PyTorch** returns the average loss across the batch (`reduction='mean'`), which is a scalar. But, if you set `reduction='none'`, then PyTorch would return the loss for each example in the batch, which would have shape `[4]` (a 1D tensor, aka a vector).

(b) Which term is called the "logits"? How do we transform logits to class probabilities?

**Solution:**  $z_{10}$  is the logits. We use softmax normalization to convert logits to probabilities.

(c) Let `batchsize=n` and let  $K$  be the total size of all intermediate activations. What is the total number of elements in all intermediate activations?

**Solution:**  $n * K$ . You can compute the total number of elements in all intermediate activations  $Z$  as follows where  $\hat{z}_i$  denotes the number of elements at  $z_i$ :

$$\begin{aligned}
 Z &= \hat{z}_1 + \hat{z}_2 + \dots + \hat{z}_{10} \\
 &= 3 * n * 128 && \text{Output of Linear1, LayerNorm, ReLU} \\
 &+ 3 * n * 256 && \text{Output of Linear2, LayerNorm, ReLU} \\
 &+ 3 * n * 20 && \text{Output of Linear3, LayerNorm, ReLU} \\
 &+ n * 1000 && \text{Output of Linear4} \\
 &= (3 * 4 * 128) + (3 * 4 * 256) + (3 * 4 * 20) + (4 * 1000) \\
 &= 8848
 \end{aligned}$$

As you can see,  $Z$  is a multiple of  $n$ .

(d) Let  $C$  be the total number of model parameters. How would you compute this? (You don't need to do the arithmetic, but you should understand how to compute  $C$  on an exam.)

**Solution:** To compute  $C$ , you would need to sum:

- The total number of parameters for each `Linear` layer, where each layer has its weights (`in_feats * out_feats`) and biases (`out_feats`).
- The total number of parameters for each `LayerNorm` layer. Recall each of these layers has  $\gamma$  and  $\beta$  learnable parameters, each of shape `[dim_feat]` which matches the `[out_feats]` of the preceding `Linear` layer.

For reference, here is the full math for this specific model architecture:

$$\begin{aligned}
 C &= (52 * 128) + 128 && \text{Linear1 weights and biases} \\
 &+ 128 + 128 && \text{LayerNorm gamma and beta parameters} \\
 &+ (128 * 256) + 256 && \text{Linear2 weights and biases} \\
 &+ 256 + 256 && \text{LayerNorm gamma and beta parameters} \\
 &+ (256 * 20) + 20 && \text{Linear3 weights and biases} \\
 &+ 20 + 20 && \text{LayerNorm gamma and beta parameters} \\
 &+ (20 * 1000) + 1000 && \text{Linear4 weights and biases} \\
 &= 66,756
 \end{aligned}$$

(e) Recall that GPU memory is a limited resource, and assume that all `ndarrays` (e.g. activations/parameters) are stored in GPU memory. When training (or inferencing) with large `batchsizes`, do activations or parameters take up more memory?

**Solution:** It depends! Most of the time (e.g. if your model parameters aren't extremely large, e.g. trillions), activation size is larger than model parameter size. But, you can imagine constructing a

model architecture where the  $C \geq n * K$  for some batchsize  $n$ . Still, in the limit, as batchsize increases, eventually  $n * K$  will surpass  $C$  - assuming you have enough GPU memory to increase the batchsize enough!

## 2. Layer Normalization

Recall that *layer normalization* (LayerNorm) is a normalization technique that normalizes across the feature dimension (e.g. the rows) of its input  $X$ , where  $X$  has shape `[batchsize, dim_feat]`:

$$\text{LayerNorm}(X) = \frac{X - \mu}{\sqrt{\sigma^2 + \epsilon}} \odot \gamma + \beta$$

where  $\mu$  and  $\sigma^2$  are the mean and variance of each **row** of  $X$  (shape `[batchsize, dim_feat]`),  $\epsilon$  is a small constant to prevent division by zero,  $\gamma$  and  $\beta$  (both shape `[dim_feat]`) are learnable parameters that scale and shift the normalized output, and  $\odot$  denotes element-wise multiplication.

(a) Perform  $\text{LayerNorm}(X)$  given the following (assume  $\epsilon = 0$  for simplicity):

$$X = \begin{bmatrix} 4 & 4 & 8 & 8 \\ 0 & 2 & 2 & 8 \\ 1 & 1 & 1 & 5 \end{bmatrix}, \quad \gamma = [1, 2, 3, 4], \quad \beta = [0, -1, 1, 0]$$

**Solution:** First, we compute the mean and variance of each row of  $X$ :

$$\begin{aligned} \mu_1 &= 6, & \sigma_1^2 &= 4 \\ \mu_2 &= 3, & \sigma_2^2 &= 9 \\ \mu_3 &= 2, & \sigma_3^2 &= 3 \end{aligned}$$

Next, we normalize each row of  $X$ :

$$\begin{aligned} \hat{X}_1 &= \frac{X_1 - \mu_1}{\sqrt{\sigma_1^2 + \epsilon}} = \frac{[4, 4, 8, 8] - 6}{\sqrt{4 + \epsilon}} = \frac{[-2, -2, 2, 2]}{2} = [-1, -1, 1, 1] \\ \hat{X}_2 &= \frac{X_2 - \mu_2}{\sqrt{\sigma_2^2 + \epsilon}} = \frac{[0, 2, 2, 8] - 3}{\sqrt{9 + \epsilon}} = \frac{[-3, -1, -1, 5]}{3} \approx [-1, -0.33, -0.33, 1.67] \\ \hat{X}_3 &= \frac{X_3 - \mu_3}{\sqrt{\sigma_3^2 + \epsilon}} = \frac{[1, 1, 1, 5] - 2}{\sqrt{3 + \epsilon}} \approx \frac{[-1, -1, -1, 3]}{1.73} \approx [-0.58, -0.58, -0.58, 1.73] \end{aligned}$$

Finally, we scale and shift the normalized output:

$$\begin{aligned} \text{LayerNorm}(X)_1 &= \hat{X}_1 \odot \gamma + \beta \\ &= [-1, -1, 1, 1] \odot [1, 2, 3, 4] + [0, -1, 1, 0] \\ &= [-1, -2, 3, 4] + [0, -1, 1, 0] \\ &= [-1, -3, 4, 4] \\ \text{LayerNorm}(X)_2 &= \hat{X}_2 \odot \gamma + \beta \\ &\approx [-1, -0.33, -0.33, 1.67] \odot [1, 2, 3, 4] + [0, -1, 1, 0] \end{aligned}$$

$$\begin{aligned}
&\approx [-1, -0.66, -0.99, 6.68] + [0, -1, 1, 0] \\
&\approx [-1, -1.66, 0.01, 6.68] \\
\text{LayerNorm}(X)_3 &= \hat{X}_3 \odot \gamma + \beta \\
&\approx [-0.58, -0.58, -0.58, 1.73] \odot [1, 2, 3, 4] + [0, -1, 1, 0] \\
&\approx [-0.58, -1.16, -1.74, 6.92] + [0, -1, 1, 0] \\
&\approx [-0.58, -2.16, -0.74, 6.92]
\end{aligned}$$

Therefore, the output of  $\text{LayerNorm}(X)$  is:

$$\text{LayerNorm}(X) = \begin{bmatrix} -1 & -3 & 4 & 4 \\ -1 & -1.66 & 0.01 & 6.68 \\ -0.58 & -2.16 & -0.74 & 6.92 \end{bmatrix}$$

- (b) Notice that in the previous subpart, `batchsize = 3`. If we instead had `batchsize = 1`, can we still take the `LayerNorm`? Why or why not?

**Solution:** Yes, because `LayerNorm` normalizes across the feature dimension (e.g. the rows) of its input  $X$ , so it is not dependent on the batch size.

### 3. Batch Normalization

Recall that *batch normalization* (`BatchNorm`) is a normalization technique that normalizes across the batch dimension (e.g. the columns) of its input  $X$ , where  $X$  has shape `[batchsize, dim_feat]`:

$$\text{BatchNorm}(X) = \frac{X - \mu}{\sqrt{\sigma^2 + \epsilon}} \odot \gamma + \beta$$

where  $\mu$  and  $\sigma^2$  are the mean and variance of each **column** of  $X$  (shape `[batchsize, dim_feat]`),  $\epsilon$  is a small constant to prevent division by zero,  $\gamma$  and  $\beta$  (both shape `[dim_feat]`) are learnable parameters that scale and shift the normalized output, and  $\odot$  denotes element-wise multiplication.

**Note that both the element-wise multiplication and sum with  $\beta$  broadcasts across the batch dimension.**

- (a) Perform  $\text{BatchNorm}(X)$  given the following (assume  $\epsilon = 0$  for simplicity):

$$X = \begin{bmatrix} 2 & 2 & 1 & 1 \\ 2 & 5 & 4 & 1 \\ 5 & 5 & 10 & 10 \end{bmatrix}, \quad \gamma = [1, 2, 3, 4], \quad \beta = [0, -1, 1, 0]$$

**Solution:** First, we compute the mean and variance of each column of  $X$ :

$$\begin{aligned}
\mu_1 &= 3, & \sigma_1^2 &= 2 \\
\mu_2 &= 4, & \sigma_2^2 &= 2 \\
\mu_3 &= 5, & \sigma_3^2 &= 14 \\
\mu_4 &= 4, & \sigma_4^2 &= 18
\end{aligned}$$

Next, we normalize each column of  $X$ :

$$\begin{aligned}\hat{X}_1 &= \frac{X_1 - \mu_1}{\sqrt{\sigma_1^2 + \epsilon}} = \frac{[2, 2, 5] - 3}{\sqrt{2 + \epsilon}} \approx [-0.71, -0.71, 1.41] \\ \hat{X}_2 &= \frac{X_2 - \mu_2}{\sqrt{\sigma_2^2 + \epsilon}} = \frac{[2, 5, 5] - 4}{\sqrt{2 + \epsilon}} \approx [-1.41, 0.71, 0.71] \\ \hat{X}_3 &= \frac{X_3 - \mu_3}{\sqrt{\sigma_3^2 + \epsilon}} = \frac{[1, 4, 10] - 5}{\sqrt{14 + \epsilon}} \approx [-1.07, -0.27, 1.34] \\ \hat{X}_4 &= \frac{X_4 - \mu_4}{\sqrt{\sigma_4^2 + \epsilon}} = \frac{[1, 1, 10] - 4}{\sqrt{18 + \epsilon}} \approx [-0.71, -0.71, 1.41]\end{aligned}$$

Finally, we scale and shift the normalized output. Recall that the element-wise multiplication and sum with  $\beta$  are broadcasted across the batch dimension, e.g.:

$$\text{BatchNorm}(X) = \hat{X} \odot \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix} + \begin{bmatrix} 0 & -1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & -1 & 1 & 0 \end{bmatrix}$$

Here is the equivalent computation for each column since we're computing by hand for this exercise:

$$\begin{aligned}\text{BatchNorm}(X)_1 &= \hat{X}_1 \odot \gamma + \beta \approx [-0.71, -0.71, 1.41] \odot [1, 1, 1] + [0, 0, 0] = [-0.71, -0.71, 1.41] \\ \text{BatchNorm}(X)_2 &= \hat{X}_2 \odot \gamma + \beta \approx [-1.41, 0.71, 0.71] \odot [2, 2, 2] + [-1, -1, -1] = [-3.82, 0.42, 0.42] \\ \text{BatchNorm}(X)_3 &= \hat{X}_3 \odot \gamma + \beta \approx [-1.07, -0.27, 1.34] \odot [3, 3, 3] + [1, 1, 1] = [-2.21, 0.19, 5.02] \\ \text{BatchNorm}(X)_4 &= \hat{X}_4 \odot \gamma + \beta \approx [-0.71, -0.71, 1.41] \odot [4, 4, 4] + [0, 0, 0] = [-2.84, -2.84, 5.64]\end{aligned}$$

Therefore, the output of  $\text{BatchNorm}(X)$  is:

$$\text{BatchNorm}(X) = \begin{bmatrix} -0.71 & -3.82 & -2.21 & -2.84 \\ -0.71 & 0.42 & 0.19 & -2.84 \\ 1.41 & 0.42 & 5.02 & 5.64 \end{bmatrix}$$

- (b) Notice that in the previous subpart, `batchsize = 3`. If we instead had `batchsize = 1`, can we still take the `BatchNorm`? Why or why not?

**Solution:** No, because `BatchNorm` normalizes across the batch dimension (e.g. the columns) of its input  $X$ , so it is dependent on the batch size. If there is only one element in the batch, we cannot compute a meaningful mean and variance.

## 4. Dropout

Recall that *dropout* is a regularization technique where each input unit has a probability  $p$  of being set to zero ("dropped out") during training to prevent overfitting. There is also a variant of dropout called *dropout with correction* where we scale up the units that aren't zeroed out by  $\frac{1}{1-p}$ .

In this problem, we'll perform dropout on a toy example. Suppose we have an input vector  $x = [2, 6, 4, 4, 5, 3, 4, 4]$ , weight vector  $w = [1, 1, 1, 1, 1, 1, 1, 1]$ , output  $y = w \cdot x$  (dot product), and dropout probability  $p = 0.75$ .

Additionally, assume that dropout results in the first 6 values being dropped (set to 0).

- (a) Compute  $y_{\text{drop}}$ , the output of regular dropout (without correction).

**Solution:**

$$\begin{aligned} y_{\text{drop}} &= w \cdot x_{\text{drop}} \\ &= 1 * 0 + 1 * 0 + 1 * 0 + 1 * 0 + 1 * 0 + 1 * 0 + 1 * 4 + 1 * 4 \\ &= 0 + 0 + 0 + 0 + 0 + 0 + 4 + 4 \\ &= 8 \end{aligned}$$

- (b) Compute  $y_{\text{corr}}$ , the output of dropout with correction.

**Solution:**

$$\begin{aligned} y_{\text{corr}} &= w \cdot x_{\text{corr}} \\ &= 1 * 0 + 1 * 0 + 1 * 0 + 1 * 0 + 1 * 0 + 1 * 0 + 1 * 4 * \left(\frac{1}{1-0.75}\right) + 1 * 4 * \left(\frac{1}{1-0.75}\right) \\ &= 0 + 0 + 0 + 0 + 0 + 0 + 1 * 4 * 4 + 1 * 4 * 4 \\ &= 16 + 16 \\ &= 32 \end{aligned}$$

- (c) Compute  $y_{\text{orig}}$ , the output without any dropout (i.e. the original output).

**Solution:**

$$\begin{aligned} y_{\text{orig}} &= w \cdot x \\ &= 1 * 2 + 1 * 6 + 1 * 4 + 1 * 4 + 1 * 5 + 1 * 3 + 1 * 4 + 1 * 4 \\ &= 2 + 6 + 4 + 4 + 5 + 3 + 4 + 4 \\ &= 32 \end{aligned}$$

- (d) Observe that  $\|y_{\text{corr}}\|$  more closely matches  $\|y_{\text{orig}}\|$  than  $\|y_{\text{drop}}\|$  does. Why is this desirable?

**Solution:** Simply zeroing out units like in  $y_{\text{drop}}$  would change the expected value of the output during training compared to test time when we disable dropout (this is called "domain shift"), which would decrease test-time task performance. By scaling up the units that aren't zeroed out by  $\frac{1}{1-p}$ , dropout with correction maintains the expected value of the output during training, resulting in better performance. (Note that it is not guaranteed that  $y_{\text{corr}}$  will always be closer to  $y_{\text{orig}}$  than  $y_{\text{drop}}$  is, but in expectation, it will be.)

- (e) Dropout is an example of a layer that has different behavior at train vs. test time. What other layer also has different behavior for train vs. test time?

**Solution:** BatchNorm. During training, BatchNorm normalizes using the mean and variance of the current batch, while during test time, it normalizes using running estimates of the mean and variance computed during training.

## 5. $\ell_2$ Regularization

Recall that  $\ell_2$  regularization is a technique used to prevent overfitting by adding a penalty term to the loss function that encourages the model parameters to be small. The  $\ell_2$  regularization term is defined as:

$$\ell_2(\theta) = \lambda \cdot \|\theta\|_2^2$$

where  $\theta$  represents the model parameters,  $\|\theta\|_2^2$  is the squared  $\ell_2$  norm of  $\theta$  (i.e. the sum of the squares of the parameters), and  $\lambda$  is a hyperparameter that controls the strength of the regularization.

Consider the following optimization problem with  $\ell_2$  regularization where  $\theta$  is the model parameters,  $\ell_{ce}$  is cross-entropy loss,  $h$  is the hypothesis function (e.g. output logits of a neural network),  $X$  is the input data, and  $y$  is the true labels:

$$\arg \min_{\theta} \ell_{ce}(h(X), y) + \lambda \cdot \|\theta\|_2^2$$

- (a) When  $\lambda$  is very large, which of the following  $\theta$  values are likely to be the result of the above optimization?
- (a)  $\theta = 0$
  - (b)  $\theta = 1$
  - (c)  $\theta = 1000000$
  - (d)  $\theta = \theta^*$ , where  $\theta^*$  is the result of solving:  $\arg \min_{\theta} \ell_{ce}(h(X), y)$

**Solution:** (a)  $\theta = 0$ . With a large  $\lambda$ , the optimization will ignore the  $\ell_{ce}$  term, and focus on minimizing  $\|\theta\|_2^2$ , which is to set  $\theta = 0$ .

- (b) When  $\lambda$  is very small (e.g.  $\lambda = 0.0000001$ ), which of the following  $\theta$  values are likely to be the result of the above optimization?
- (a)  $\theta = 0$
  - (b)  $\theta = 1$
  - (c)  $\theta = 1000000$
  - (d)  $\theta = \theta^*$ , where  $\theta^*$  is the result of solving:  $\arg \min_{\theta} \ell_{ce}(h(X), y)$

**Solution:** (d)  $\theta = \theta^*$ . With a small  $\lambda$ , the optimization will ignore the regularization term, and focus on minimizing  $\ell_{ce}(h(X), y)$ , which is to set  $\theta = \theta^*$ .

- (c) Suppose we solve  $\theta^* = \arg \min_{\theta} \ell_{ce}(h(X), y)$ . Suppose we then add an  $\ell_2$  regularization term with a "moderate" value of  $\lambda$  such that  $\theta = \arg \min_{\theta} \ell_{ce}(h(X), y) + \lambda \cdot \|\theta\|_2^2$ .
- i. How would you expect  $\|\theta\|_2$  to compare against  $\|\theta^*\|_2$ ?
 

**Solution:** We would expect  $\|\theta\|_2 < \|\theta^*\|_2$ . This is because the regularization term encourages smaller parameter values, so the optimization will find a  $\theta$  that has a smaller  $\ell_2$  norm than  $\theta^*$ , which only minimizes the  $\ell_{ce}$  term.
  - ii. How would you expect  $\theta$  to compare against  $\theta^*$  when comparing **training** dataset metrics (e.g. **classification accuracy**)?
 

**Solution:** We'd expect  $\theta$  to behave a little worse on training dataset metrics than  $\theta^*$ . This is because the presence of the regularization term introduces a tradeoff between "do well on the training dataset" and "satisfy the regularization penalty".

- iii. How would you expect  $\theta$  to compare against  $\theta^*$  when comparing **test** dataset metrics (e.g. **generalizability**)?

**Solution:** We'd hope that  $\theta$  generalizes better to unseen data than  $\theta^*$  due to the presence of the added regularization term. This is the motivation for why we do regularization in the first place: to train models that don't overfit on the training data, and generalize better to unseen data.

**Contributors:**

- Eric Kim.
- Rebecca Dang.