CS 282 Deep Neural Networks **Fall 2022**

Lecture 9: ConvNets/CV

Lecturer: Anant Sahai *Scribe: Buyu Zhang, Michael Lam*

9.1 Input standardization, Normalization

A common maxim in machine learning is that input data should "always" be normalized, meaning the input data should somehow follow a distribution with zero mean and unit variance, i.e. a transformation of the form $x \to \frac{x - \text{“}E[X]\text{”}}{\text{“}\sigma[X]\text{”}}$ where subtraction by the expected value "E[X]" zeroes the mean, and division by the standard deviation "$\sigma[X]$" unitizes the spread. But this normalization raises several questions. Why are we even normalizing in the first place? Where do "E[X]" and, by extension, "$\sigma[X]$" come from?

9.1.1 Reasons for Normalization

Aside: Confidence in training points
Before we delve into ways to normalize our data, let's look at a toy example that demonstrates the motivation for normalization. Consider the following standard least squares problem:

$$X\overrightarrow{w} \approx \overrightarrow{y} \tag{9.1}$$

Suppose we know measurement $y[2]$ to a greater degree of confidence, i.e.

$$y[2] = y_{true}[2] + N(0, 0.01), \tag{9.2}$$

while other data points are described by

$$y[i] = y_{true}[i] + N(0, 1), i \neq 2 \tag{9.3}$$

How can we modify our ordinary least-squares algorithm to take advantage of this information? Intuitively, we'd like to somehow weight $y[2]$ more heavily since it carries more "information", in a sense, because we're more confident that it is close to the true value.

To do this, notice that the variance of y[2] is 0.01 times that of all the rest of the data points. If we multiply the second data point and its corresponding entries in X by 10, the variance of y[2] will now equal one (keep in mind that variance scales with the square of the data point's scaling factor). Not only that, but our second training point will now be weighted 10 times more than it was before.

$$10 \times y[2] = 10 \times y_{true}[2] + 10 \times N(0, 0.01) = 10 \times y_{true}[2] + N(0, 1) \tag{9.4}$$

Solution: Multiply second row of X by 10, second row of y by 10, do standard least squares.

Things to consider: What if we instead duplicate the second data point and its corresponding entries in X 10 times? How many times must we duplicate the point for the effect to be the same as scaling by 10? Is this effective when the feature space is large? (In general the effect of duplicating a datapoint will be dependent on the total number of datapoints. Duplicating a datapoint is a better choice for a non-linear model.)

We demonstrated in the above aside that it was possible to preprocess the data such that a specific data point which we know with high confidence can be weighted proportionally to its spread. Note that, in doing so, the data point now has a mean of zero and unit variance. *But that's exactly what normalization is!*

In the toy example above, normalization had the effect of correctly weighting points based on their importance (in this case, confidence). Normalizing our training data before inputting them into neural networks has a similar effect of scaling the parameters such that the raw *magnitudes* of the values don't necessarily impact the gradient calculation during the gradient descent backpropagating step (if we had massive, un-normalized values, the gradient would always be large). Moreover, the data points are generally centered at zero because the *ReLU* layers are often initialized at that point, making the neural net most sensitive and expressive when input values are around zero. Thus, normalization can additionally decrease the neural network's training time.

Food for thought: We often normalize our data beforehand because we don't expect the *magnitude* of the values to have any value in the classification process, but this isn't always true. Can you think of an example where normalization would preclude accurate classification?

9.1.2 Choices for "$E[X]$"

We are not generally provided with prior knowledge of the probability distribution from which our sample X data are drawn, so these values must be derived empirically, i.e. from the data set X itself during training time. Let us focus firstly on ways such an empirical average $E[X]$ can be derived.

Consider the following scenario where we have n 100 x 100 images in our data set, each with 3 channels: red, green and blue, with values between 0 and 255, inclusive. The following are ways we can compute an empirical expectation for any given point in any image:

1. **For any given position (x, y) in channel i, take the expected value corresponding to that point to be the average of the channel i values at (x, y) across all images**, i.e. the expected value for position (50, 50) in the red channel is the average of the red values at (50, 50) across all images. This can be done across our entire data set, with or without the addition of augmented images.
2. **Use the numeric midpoint of the range as the expected value for all positions** (e.g. choose 128 for range 0-255). This method is simple, but it assumes that the color values across all images are distributed evenly around the middle of the range. Consider what might happen if our data set consisted only of dark images (images with channel values between, say, 0 and 12). Would our normalization still be centered at 0?
3. **Take the expected value for position (x, y) at channel i to be the average of all the channel i values for that image.** For example, for any given image, the expected value for position (50, 50) of the red channel would be the average of the red channel value across all positions of that specific image.
4. **Set the expected value to be an average of the values within a local patch of pixel positions in the same channel within the same image**. For the expected value for position (50, 50) in the red channel, we may want to take the average of a 3×3 patch centered at that coordinate, i.e. the rectangle $[49, 51] \times [49, 51]$.

Food for thought: What are some of the advantages/disadvantages of some of the expected value definitions listed above? Especially for the definitions that average within a single image, what information may be lost in the normalization process?

Note: You may be wondering whether normalization may change the input in a way that reduces the information it contains and derails the optimization process. The good news is that our normalization, defined as $\frac{x - \text{“}E[X]\text{”}}{\text{“}\sigma[X]\text{”}}$, is composed of subtraction and division functions that can be undone by our biases and weights, respectively. Keep in mind that there are important exceptions given how our convolutional networks are structured. For example, due to the moving nature of the filter, there is no way for an expected value defined as the average of a particular position across all images (option 1 listed above) to be reversed. However, an average taken across all positions in all images would be reversible. Can you come up with any other examples?

These are just a few of the more common ways to compute an empirical expected value. Note that the empirical variance is calculated analogously, except, instead of averaging, we take the variance across a certain subset of points. There are myriad other definitions depending on the specific application, some of which are combinations/variations of the ones listed above. For example, for a much coarser average, we could define $E[X]$ to be the average of all values within the image across all channels (a variation of option 1). One could also imagine averaging across all channel positions across all images as an extension of this definition. When choosing what type of normalization to use for a specific problem, it is considered good practice to use whatever method has given good results for a similar problem in the past.

Each expectation calculation scheme can be seen as different ways to span 3 axes: the image positions, the image channels, and the different images within the data set (Figures 9.1-9.4). For example, we can visualize expectation (3), the average of the values at all positions in one channel for one image, as a prism that spans all positions, 1 channel, and 1 image (Figure 9.1). A few other examples are listed in Figures 9.1-9.4.

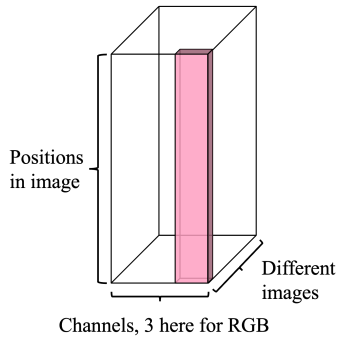


Figure 9.1: Demonstration for option 3.

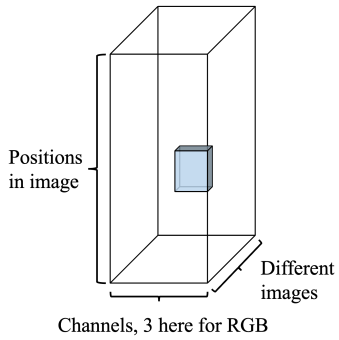


Figure 9.2: Demonstration for option 4.

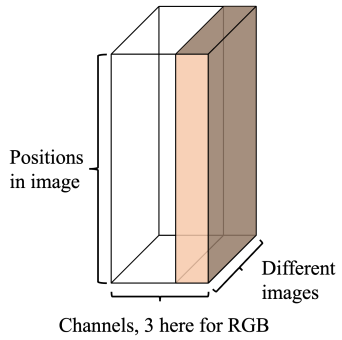


Figure 9.3: Variation of option 3.
(Averaging all images)

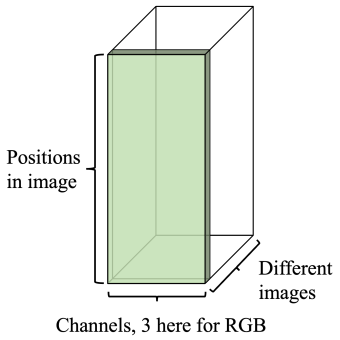


Figure 9.4: Variation of option 3.
(Averaging all channels)

9.1.3 Normalization methods

Some of the more common normalization methods have special names:

- **Batch normalization**
  Batch normalization normalizes the contributions to a layer for every mini-batch.
  At training time, the gradients are not calculated for all data at one time; instead, a batch of data is used. A batch normalization layer uses a mini-batch of data to estimate the mean and standard deviation of each feature. These estimated means and standard deviations are then used to center and normalize the features of the mini-batch. A running average of these means and standard deviations is kept during training, and at test time these running averages are used to center and normalize features. The batch normalization will become unstable when the batch size is too small.

- **Layer normalization**
  Layer normalization normalizes input across all channels in one image.

- **Instance normalization**
  Instance normalization normalizes across each channel in each training images. The problem instance normalization tries to address is that the network should be agnostic to the contrast of the original image.

- **Group normalization**
  Group Normalization normalizes over a group of channels for each training images.
  Group normalization is a medium between Instance normalization and layer normalization. When we put all the channels into a single group, group normalization becomes layer normalization. When we consider each individual channel a single group, it becomes instance normalization.

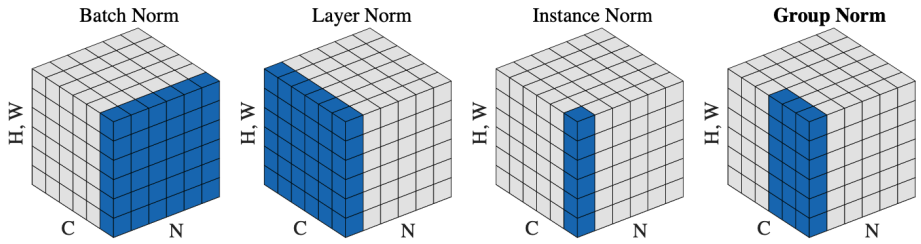


Figure 9.5: **Normalization methods.** Each subplot shows a feature map tensor, with N as the batch axis, C as the channel axis, and (H, W) as the spatial axes. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels.[1]

9.1.4 Weight Standardization

What we talked before is input standardization. Another way to control the movement of gradient descent is weight standardization. Instead of applying the weights directly during the gradient calculation, the weights are normalized beforehand, preventing weights from getting too big.

9.1.5 Deep Net Structure for Convolutional Neural Nets

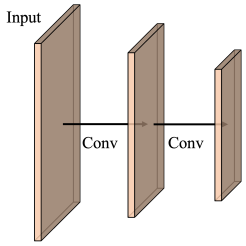


Figure 9.6: Deep net structure for Conv-nets

We've established that normalization in the input layer centers the data where the ReLU elbows are most active, but do we need to normalize each layer in the convolutional net (Figure 9.5)? After all, the output of each layer is the input to the next layer, and the repeated applications of the weighted convolutional layers can very quickly lead to an output that is, again, extremely small (leading to a problem known as the *vanishing* gradient) or very large (the *exploding* gradient).

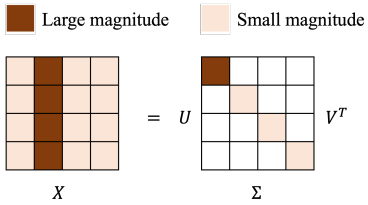
The solution is to not just normalize the initial training data, but to also normalize the outputs of intermediate convolutional layers. Intuitively, normalizing after every layer in the network should solve the problem, which was exactly what early researchers placed did. But since gradients don't tend to get extremely large or small over the course of the application of at least a few weighted convolutional layers, adding normalization at the very start and after every few layers is sufficient and is what is typically used in modern networks.

Aside: Large singular values
Consider the singular value decomposition of an arbitrary output matrix X:

$$X = U\Sigma V^T \tag{9.5}$$

The diagonal entries of Σ are the singular values of the matrix X, which is incidentally related the X's Frobenius norm, a measure of matrix's "mass", or the magnitude of its entries. There are two ways these singular values can get very large:

1. **A few columns of the X have extremely large magnitude.** The corresponding singular values for these columns would then be correspondingly large. This is the case where a few data points are much larger than all the others.



2. **All the columns are close to being collinear.** If many of the data points fall on the same line, the singular value corresponding to that direction will be large. This is the case when most data points follow a certain trend.

Normalization helps to ensure that large singular values come not from the former case, but from the latter. Gradient descent is thus more robust against large, outlying data points and more sensitive to strong trends.

9.2 Residual network

Another way to combat the effects of the vanishing/exploding gradients is the introduction of *skip connections*, bridges that allow the output in one layer to be the input to not just the layer immediately subsequent, but also to layers further down the net. In a network without skip connections, gradient updates must pass through all subsequent layers before reaching the weights of the current layer, which may lead to a gradient update that is minuscule. Skip connections make the weight layers more sensitive to gradient updates as the gradients have to backpropagate through much fewer layers of the network during the gradient calculation and are therefore much less likely to get extremely small from repeated applications of weight layers. Figure 9.7 contrasts the structure of the plain net with that of a residual net.

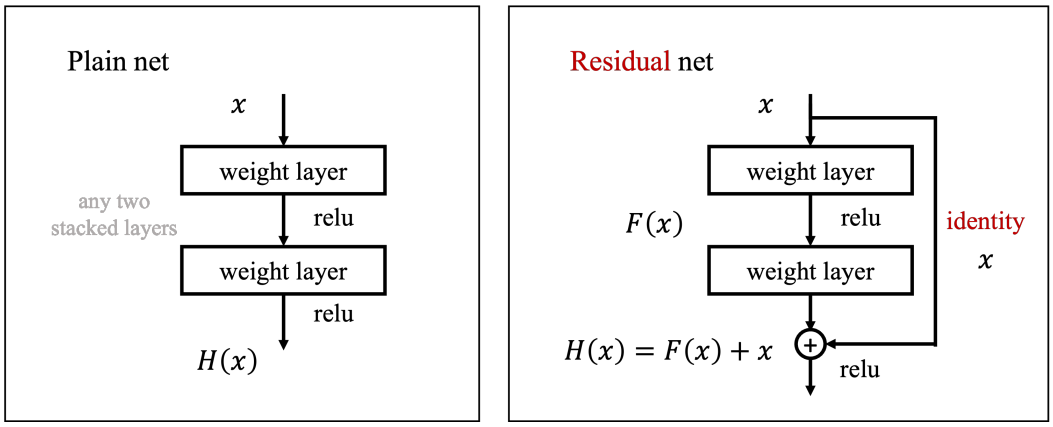


Figure 9.7: Plain net (left) and residual net (right).[2]

References

[1] Yuxing Wu, Kaiming He, Group Normalization, *Proceedings of the European conference on computer vision (ECCV)* (2018).

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren,Jian Sun, Deep Residual Learning for Image Recognition, *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016).