Deep Neural Networks **(UC Berkeley CS182/282A, Spring 2023)**

Lecture 9: Batch norm, Dropout, Data Augm and ResNet

17 February 2023

Lecturer: Anant Sahai *Scribes: Margarita Geleta and Jerry Sun*

1 Batch Normalization

Recall the standard ML approach for data normalization: given a dataset of pairs of labeled data points $\mathcal{D} = (x_i, y_i)_{i=1,...,N}$, it is pre-processed such that the new normalized dataset $\tilde{\mathcal{D}}$ is (Equation 1):

$$\begin{aligned} \tilde{\mathcal{D}} = (\tilde{x}_i, y_i)_{i=1,...,N} &= \left(\frac{x_i - \mu}{\sigma}, y_i\right)_{i=1,...,N} \\ \text{where } \mu &= \frac{1}{N}\sum_{i=1}^{N} x_i \\ \text{and } \sigma^2 &= \frac{1}{N-1}\sum_{i=1}^{N}(x_i - \mu)^2 \end{aligned} \tag{1}$$

This new dataset $\tilde{\mathcal{D}} = (\tilde{x}_i, y_i)$ is used for training. When we make this transformation for training and suppose we learn a linear classifier, we will learn something that will resemble our training set. In other words, our new test samples will be expected to be in the same range as the normalized data. Thus, the transformation applied to x_i in this process will become part of the model as well. In vector representation, where $\boldsymbol{x}_i \in \mathbb{R}^d$ and $\boldsymbol{y}_i \in \mathbb{R}^q, i = 1, ..., N$, with the operations being applied element-wise, the normalized training dataset becomes (Equation 2):

$$\begin{aligned} \tilde{\mathcal{D}} = (\tilde{\boldsymbol{x}}_i, \boldsymbol{y}_i)_{i=1,...,N} &= \left(\frac{\boldsymbol{x_i} - \boldsymbol{\mu}}{\boldsymbol{\sigma}}, \boldsymbol{y}_i\right)_{i=1,...,N} \\ \text{with } \boldsymbol{\mu} &= \frac{1}{N}\sum_{i=1}^{N} \boldsymbol{x}_i \\ \text{and } \boldsymbol{\sigma}^2 &= \frac{1}{N-1}\sum_{i=1}^{N}(\boldsymbol{x}_i - \boldsymbol{\mu})^2 \end{aligned} \tag{2}$$

For a new test point $\boldsymbol{x}$, the transformation procedure would make use of and remember the $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ computed on the training data (Equation 3):

$$\tilde{\boldsymbol{x}} = \frac{\boldsymbol{x} - \boldsymbol{\mu}}{\boldsymbol{\sigma}} \tag{3}$$

In contrast to the standard ML approach, we aim to apply normalization at every layer of the network. Additionally, in the case of the absence of access to all the N data points

(i.e., because of computational and memory constraints), *batch normalization* computes the normalization transformation using only a batch $\mathcal{B}$ instead of the whole dataset $\mathcal{D}$. Therefore, because we are only using the data points in the current batch to compute the mean and standard deviation, each batch $\mathcal{B}_k$ will have its own pair of $\boldsymbol{\mu}_k$ and $\boldsymbol{\sigma}_k$, which will be different from the mean and standard deviation of any other batch, giving rise to the first complication with batch normalization:

Problem 1: We have different $\boldsymbol{\mu}_k$ and $\boldsymbol{\sigma}_k$ for every batch $\mathcal{B}_k$. A natural question to ask is, given a single new test data point, *which pair* $(\boldsymbol{\mu}_k, \boldsymbol{\sigma}_k)$ *should we use?*

We can proceed with the naive idea. For a new test point $\boldsymbol{x}$, use the mean of the batch means $\boldsymbol{\mu}_k$ and standard deviations $\boldsymbol{\sigma}_k$. Considering batch size N_B and the number of batches being $K = \lceil NN_B \rceil$ (Equation 4):

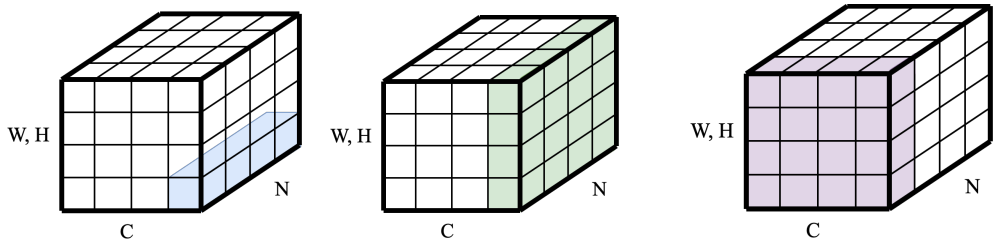
$$\tilde{\boldsymbol{x}} = \frac{\boldsymbol{x} - \frac{1}{K}\sum_{k=1}^{K} \boldsymbol{\mu}_k}{\frac{1}{K}\sum_{k=1}^{K} \boldsymbol{\sigma}_k} \tag{4}$$

When we train using batches, assume we use SGD for the optimization process. However, a problem arises because there are multiple μ and σ from our many training epochs. This is because, as training evolves, the means and standard deviations at every single layer also change, a problem known as *covariate shift* (i.e., the input distribution changes between the training and testing phases). Thus, when we say to take the average of the means and standard deviations it is also not clear from what epoch should we use them. There are a couple of different existing solutions, the easiest one is to take the means and standard deviations of the last epoch (when it converges, call it the t-th epoch) or by running the network on the training data one last time and compute the parameters (Equation 5).

$$\mathrm{BN}(\boldsymbol{x}) = \frac{\boldsymbol{x} - \boldsymbol{\mu}^{(t)}}{\boldsymbol{\sigma}^{(t)}} \tag{5}$$

Problem 2: Suppose our input data points have a high dimensionality ($\gg d$), *what do we average together?*

To illustrate, imagine our data points represent images of size $W \times H$. We arrange the batch data points into a box of size $N_B \times C \times (W \cdot H)$ (Figure 1), where N_B is the number of samples in the batch, C is the number of channels in the image (e.g., $C = 3$ for RGB images), and $(W \cdot H)$ is the number of different pixels in the image.
In the standard ML perspective (Figure 1a) we would take a particular pixel on a particular channel and average it across the batch. But since we do not have enough things in a batch, we need to average across more things. And we should average across things that are alike. In the case of batch normalization (Figure 1b) it is considered that the different pixels are all alike each other, but the different channels can be qualitatively different because they correspond to different features. So we average across the different pixels. Another kind of normalization also used in practice is *layer normalization* (Figure 1c) where the different channels are averaged together. This can be done more frequently at later layers



(a) Average in standard ML approach. (b) Average in batch normalization. (c) Average in layer normalization.

Figure 1: Source: own figure.

in the convolutional process because these features become learned features and there is no reason for them to be much larger than the others. This gives the advantage of a lot more averaging and removes the need to look deeper into your batch. Additionally, layer norm is computationally less costly than batch norm and in practice useful if we have hardware constraints. There are also variations of layer normalizations, such as *group normalization*, which consists in averaging groups of channels – average only certain channels together, channels that make sense to be averaged together.

Problem 3: We do not always want to shift our layer input distributions towards zero-mean and unit-variance. So, *what if we do not want batch normalization to force zero-mean or unit-variance?*

To solve this, we can add trainable parameters γ and β to the normalization layer which will allow us to have different means and different standard deviations (Equation 6).

$$\mathrm{BN}_{\gamma,\beta}(\boldsymbol{x}) = \gamma \frac{\boldsymbol{x} - \boldsymbol{\mu}}{\boldsymbol{\sigma}} + \beta \tag{6}$$

2 Dropout

We cannot use batch normalization in the context of fully-connected layers. Can we induce our model during training such that the interesting things that are learned are correlations across many things and not just one? Dropout is a way of making that happen. During training, dropout randomly zeroes out the outputs of neurons, i.e., certain activations are just zeroed out and they will no longer contribute to the next layer. That means that during back-propagation no gradients will pass through it because it has been clamped to zero. The idea for doing this is that there are two ways of learning a function. The first is to have one unit capturing one thing, and have another unit capturing another, which is intuitive for interpretability. The second is what dropout employs – to have a mixture of both.

Problem 1: In batch normalization training is different from testing. Likewise, the same situation happens with dropouts. *How do we apply dropout on test data points?*

A basic thought can be: since we trained with dropout, we should test with dropout as well. But this induces non-deterministic behavior in the output which is not very desirable. The standard approach is to simulate the *expected behavior* of the dropout during test time. That is why we should consider that the weights during training are bigger than during test time (if we do a dropout of a factor of α, we later divide by α). *PyTorch*'s implementation is slightly different: instead of using the expected behavior of the dropout at test time and treating test time as a special case, they reverse it and treat training time as a special case. So it treats all the weights during training time as if they were on a larger scale than they would be otherwise.

Question 1: *Can we (and should we) use dropout in convolutions?*

If we randomly zero out pixels, we are, essentially, changing the shape of the image and this might be not a desirable behavior in the learned filters. *N.B.* There is a kind of dropout that is connected to ResNets which is called *Stochastic Depth Regularization* and it drops entire blobs of elements.

2.1 Classical Motivation for Dropout

If you think about a random forest (and ensemble methods), you inject randomness so that different trees are different from each other and we learn different trees and count on the idea that by averaging lots of trees we would be cutting out the local variations and finding the pattern. Dropout can be seen as a weird twist on ensemble methods – instead of learning one neural network, we can learn a bunch of them. So how exactly do we generate a group of a neural network? Just as in a random forest, to get diversity, we lock several features from being used. With dropout the lock operation is the zeroing out of an operation – when we zero out a feature, we just disallow using it in the training. Think of this as an ensemble where each model shares weights (to avoid computational burden), but each model uses a subset of features.

2.2 Singular Values Motivation

Dropout can be motivated by a singular values approach. We know that gradient descent will respond more to the larger singular values (either correlational large, or large in some components). Although standard ML considers that largeness may not mean a lot, and can attribute largeness to differences in units or measurements, dropout can have an important impact on large singular values. From the previous section, we know that dropout randomly zeros a fraction of the elements in a weight matrix during training. The stochastic nature of dropout reduces the influence of these high-singular-value components, which can prevent over-fitting by relying too heavily on a few patterns in the data.

3 Data Augmentation

Data augmentations have a regularizing effect – one never has enough data. The general belief is that in the best case all of the features will be represented and any data that should not be included will be supported by data telling us not to include it. We attempt to build in things that we should not depend on such that our network will not depend on that in the future. We may employ basic transformations for data augmentation, such as rotations, translations, auto-contrast, equalization, and solarization. These transformations and additional data points will force the network to be invariant to small insignificant changes. Furthermore, performing more aggressive transformations will give more robustness to the model. (Figure 2).

Figure 2: Different data augmentation methods. Hendrycks, et al. CVPR 2022.

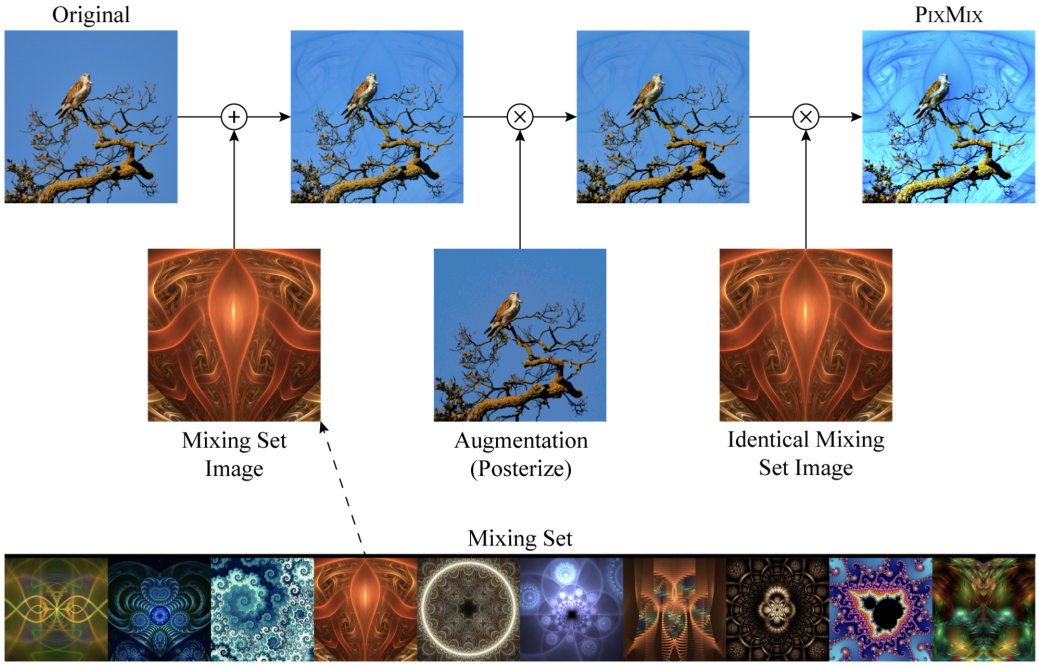


Figure 3: Mixing procedure of PixMix. Hendrycks, et al. CVPR 2022.

Psychedelic data augmentation is a type of data augmentation that transforms images a way

by adding noise in the form of visual effects such as color shifts, or visual effects. This is done in an attempt to force the neural network to learn more robust and invariant representations of the data. (Figure 3).

Label smoothing is a regularization technique used to prevent the overconfidence of models in their predictions by introducing a small amount of uncertainty into the labels. In data augmentation, label smoothing mitigates the noise of augmentations. By assigning less probability to the labels (from $(1, 0, 0)$ to $(0.9, 0.05, 0.05)$) and introducing uncertainty, the model is encouraged to handle variations in the data. Using a combination of label smoothing and psychedelic data-augmentation can lead to more robust models.

CutMix and MixUp are both data augmentation techniques that differ in how they combine different images to create new training examples. MixUp linearly interpolates their **pixel** values by taking a convex combination of the labels of the two original images. CutMix, on the other hand, randomly selects a **patch** from one image and replaces it with a patch from another image. The label for the resulting mixed image is with a weighted average of the two original images, where the weights are determined by the **areas** of the patches that were cut and pasted. CutMix can be seen as a more extreme version of MixUp, which can help deal with classes with high overlap.

4 ResNet

The most intuitive level of thinking about ResNets is to consider an image in the dataset for which one layer or block was enough to distill the relevant features for classification. However, the deeper this network is constructed, this very simple idea will now be scrambled through many different layers and may make it difficult to train this first image. Instead, if we had the output of the first layer go all the way to the last layer, then we would get gradients back that would let it train and allow for classification. If a feature is useful for classifying one feature, it could be extended to train other features. This is the basis for ResNets, which are structures that have the output of the block be the input to all of the proceeding layers, instead of just the next layers. This enables back-propagation to have multiple options that can make our gradient larger.

From the left figure, we can see that the deeper networks were not optimized as an increase in the number of layers actually led to a higher training and testing error (Figure 4). The right figure shows the improvements with ResNets. The solution that was proposed was to ensure that the gradients were hitting every block and that gradients could pass all the way back.

ResNets, also change the structure of the function F. The standard structure is a composition of functions. It is hierarchical and intuitive but in the event of a training error will mess up the gradients for the entire function.

$$F(G(H(I(\cdot)))) \tag{7}$$

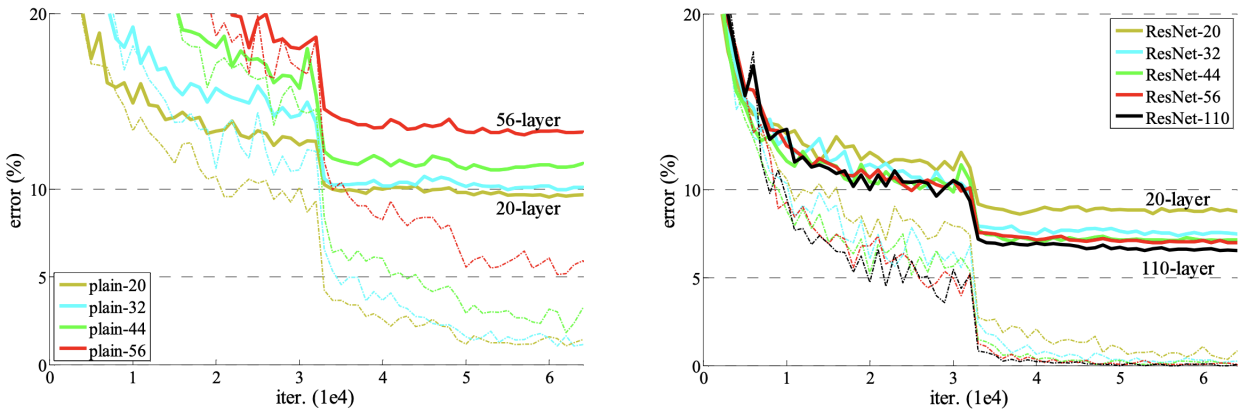
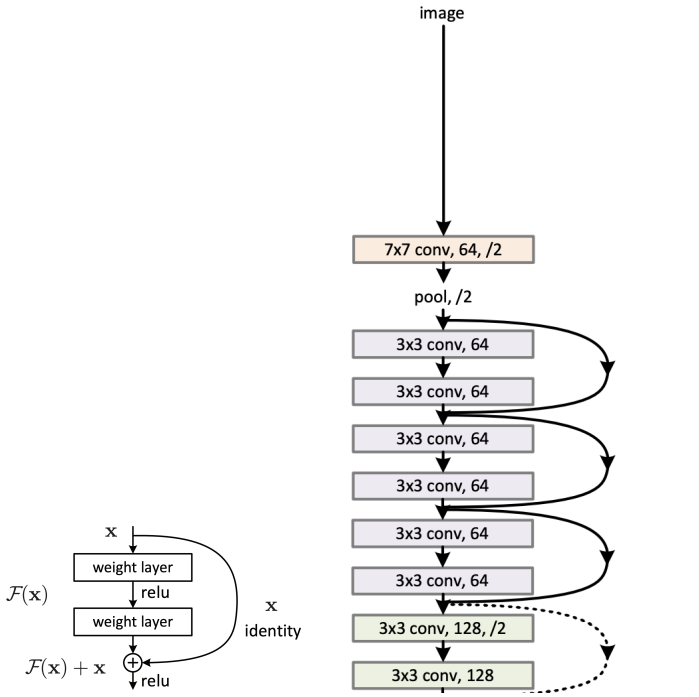


Figure 4: CIFAR-10 Experiments He, et al.

ResNets use a different sum structure that if dealing with smaller cases, can eventually resemble ordinary differential equations and can have wide applications in physics simulations (Figure 5a).

$$x + F(x) + G(F(x)) + H(G(F(x))) \tag{8}$$



(a) ResNet Structure (b) ResNet Example

Figure 5: Examples from He, et al.

5 What we wish this lecture also had to make things clearer?

1. We would like the stochastic depth regularization connection with dropouts and convolution neural networks to be better explained.

2. We would like to have the motivating example of singular values and dropout be better reinforced.

3. Explain layer normalization in more depth.

References

[1] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Netw.*, 2(5):359–366, jul 1989.