1 Batch normalization

In this discussion, we will learn what is Batch Normalization, why it is needed, and how it works. Batch Normalization was first introduced by two researchers at Google, Sergey Ioffe and Christian Szegedy in their paper 'Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift' in 2015. The authors showed that batch normalization improved the top result of ImageNet (2014) by a significant margin using only 7% of the training steps. Today, Batch Normalization is used in almost all CNN architectures.

1.1 Internal covariate shift

Before we jump into the nitty-gritty of batch normalization, let us first understand a basic principle in machine learning.

Let's say we have a flowers dataset and we want to build a binary classifier for roses. The output is 1 if the image is that of a rose and the output is 0 otherwise.

Consider a subset of the training data that primarily has red rose buds as rose examples and wildflowers as non-rose examples. These are shown in Figure 1.

Figure 1: Red rose buds

Consider another subset, shown in Figure 2, that has fully blown roses of different colors as rose examples and other non-rose flowers in the picture as non-rose examples.

Intuitively, it makes sense that every mini-batch used in the training process should have the same distribution. In other words, a mini-batch should not have only images from one of the two subsets above. It should have images randomly selected from both subsets in each mini-batch.

The same intuition is graphically depicted in Figure 3. The last column of Figure 3 shows the two classes (roses and non-roses) in the feature space (shown in two dimensions for ease of visualization). The blue curve shows the decision boundary. We can see the two subsets lie in different

Figure 2: Different color roses

regions of the feature space. This difference in distribution is called the covariate shift. When the mini-batches have images uniformly sampled from the entire distribution, there is negligible covariate shift. However, when the mini-batches are sampled from only one of the two subsets shown in Figure 1 and Figure 2, there is a significant covariate shift. This makes the training of the rose vs non-rose classifier very slow.

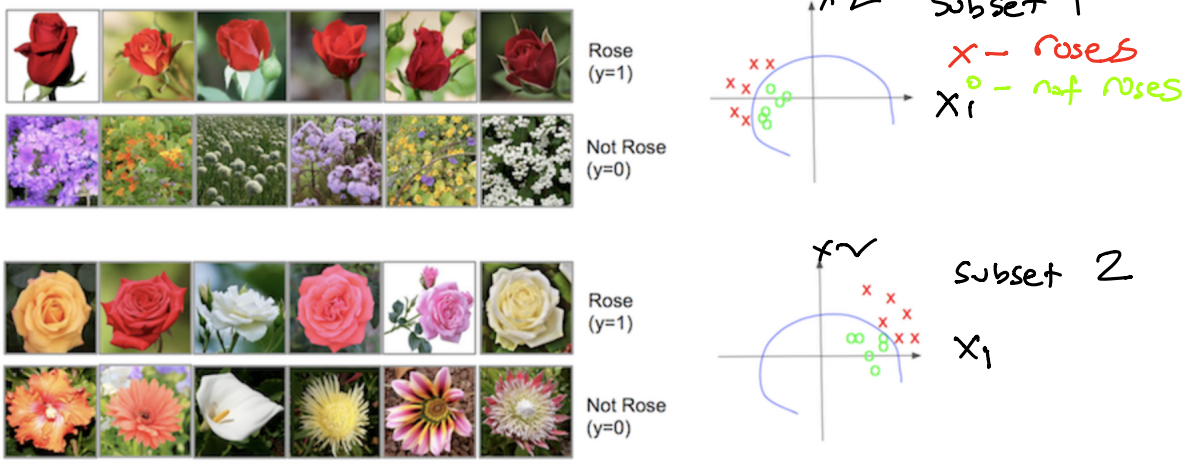


Figure 3: The top two rows of flowers show a subset of the data and the bottom two rows show a different subset of the data. The two subsets have very different distributions. The last column shows the distribution of the two classes in the feature space using red and green dots. The blue line show the decision boundary between the two classes.

An easy way to solve this problem for the input layer is to randomize the data before creating mini-batches.

But, how do we solve this for the hidden layers? Just as it made intuitive sense to have a uniform distribution for the input layer, it is advantageous to have the same input distribution for each hidden unit over time while training. But in a neural network, each hidden unit's input distribution changes every time there is a parameter update in the previous layer. This is called **internal covariate shift**. This makes training slow and requires a very small learning rate and a good parameter initialization. This problem is solved by normalizing the layer's inputs over a mini-batch

and this process is therefore called **Batch Normalization**.

By now we have an intuitive sense for why Batch Normalization is a good idea. Now, let's figure out how to do this normalization.

1.2 Batch Normalization in a Neural network

Formally, denoting $\mathbf{x} \in \mathcal{B}$ an input to Batch Normalization (BN) that is from a minibatch $\mathcal{B}$, BN transforms $\mathbf{x}$ according to the following expression:

$$BN(\mathbf{x}) = \gamma \odot \frac{\mathbf{x} - \mu_{\mathcal{B}}}{\sigma_{\mathcal{B}}} + \beta$$

where $\mu_{\mathcal{B}}$ and $\sigma_{\mathcal{B}}$ are the sample mean and sample variance of the minibatch $\mathcal{B}$. Note that γ and β are parameters that need to be learned jointly with the other model parameters.

Note that we add a small constant $\epsilon > 0$ to the variance estimate to ensure that we never attempt division by zero, even in cases where the empirical variance estimate might vanish. The estimates $\mu_{\mathcal{B}}$ and $\sigma_{\mathcal{B}}$ counteract the scaling issue by using noisy estimates of mean and variance. You might think that this noisiness should be a problem. As it turns out, this is actually beneficial. This turns out to be a recurring theme in deep learning. For reasons that are not yet well-characterized theoretically, various sources of noise in optimization often lead to faster training and less overfitting: this variation appears to act as a form of regularization.

1.3 Other benefits of batch normalization

1.3.1 Higher learning rate

If we use a high learning rate in a traditional neural network, then the gradients could explode or vanish. Large learning rates can scale the parameters which could amplify the gradients, thus leading to an explosion. But if we do batch normalization, small changes in parameter to one layer do not get propagated to other layers. This makes it possible to use higher learning rates for the optimizers, which otherwise would not have been possible. It also makes gradient propagation in the network more stable.

1.3.2 Adds regularization

Since the normalization step sees all the training examples in the mini-batch together, it brings in a regularization effect with it. If batch normalization is performed through the network, then the dropout regularization could be dropped or reduced in strength.

1.3.3 Makes it possible to use saturating non-linearities

In traditional deep neural networks, it is very hard to use saturating activation functions like the sigmoid, where the gradient is close to zero for inputs outside the range (-1,1). The change in parameters in the previous layers can easily throw the activation function's inputs close to these saturating regions and the gradients could vanish. This problem is worse in deeper networks. On

the other hand, if the distribution of the inputs to these nonlinearities remain stable, then it would save the optimizer from getting stuck in the saturated regions and thus training would be faster.

1.4 CIFAR-10 classification

Both the versions have roughly the same run time needed for each epoch (24-25 s), but as we see in Figure 4, the accuracy attains a much higher value much faster if we use the batch normalization. Without batch normalization, the model attained an accuracy of 79% after 50 epochs and with batch normalization it increased to 87% with a much faster convergence.

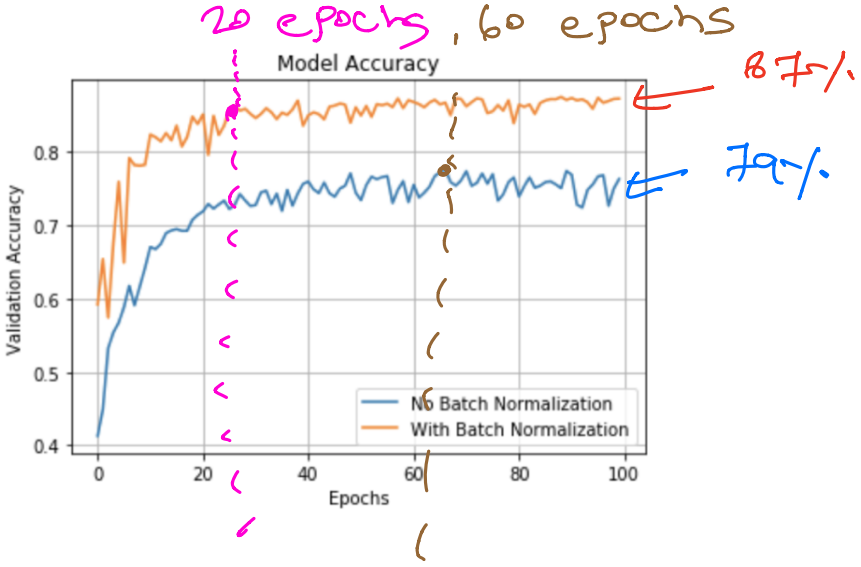


Figure 4: Model accuracy for CIFAR-10

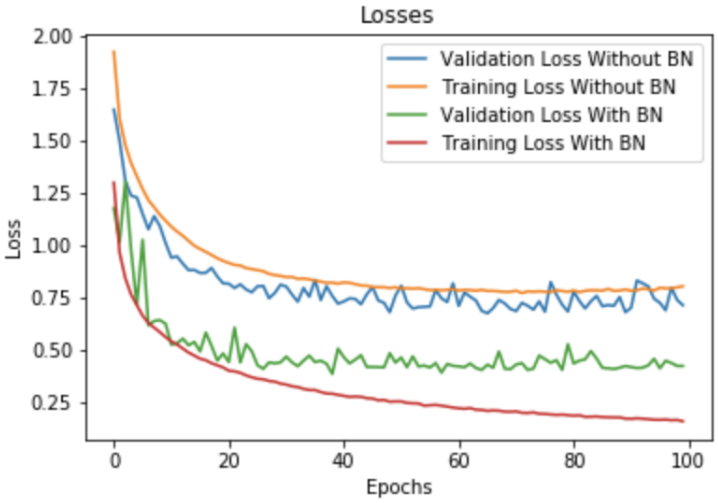


Figure 5: Loss trajectory for CIFAR-10

In Figure 5, we plotted the validation vs training losses for both the versions. As we can see the validation loss reduces substantially with batch normalization.

2 Multiple choice questions on neural networks

1. In which of the following applications can we use deep learning to solve the problem?

   (a) Protein structure prediction
   (b) Prediction of chemical reactions
   (c) Detection of exotic particles
   (d) All of these

   Since NN is a universal function approximator

2. Statement 1: It is possible to train a network well by initializing all the weights as 0
   Statement 2: It is possible to train a network well by initializing biases as 0

   (a) Statement 1 is true while Statement 2 is false
   (b) Statement 2 is true while statement 1 is false
   (c) Both statements are true
   (d) Both statements are false

   Statement 1 is false because all activations will be 0 and hence no learning.

3. The number of nodes in the input layer is 10 and the hidden layer is 5. The maximum number of connections from the input layer to the hidden layer are

   (a) 50
   (b) Less than 50
   (c) More than 50
   (d) It is an arbitrary value

   $10 \times 5 = 50$

4. Which of the following functions can be used as an activation function in the output layer if we wish to predict the probabilities of n classes $(p_1, p_2, \ldots, p_n)$ such that sum of p over all n equals to 1?

   (a) Softmax
   (b) Relu
   (c) Sigmoid
   (d) Tanh

5. Which of following activation function can't be used at output layer to classify an image?

   (a) Sigmoid
   (b) Tanh
   (c) Relu

(d) None of the above

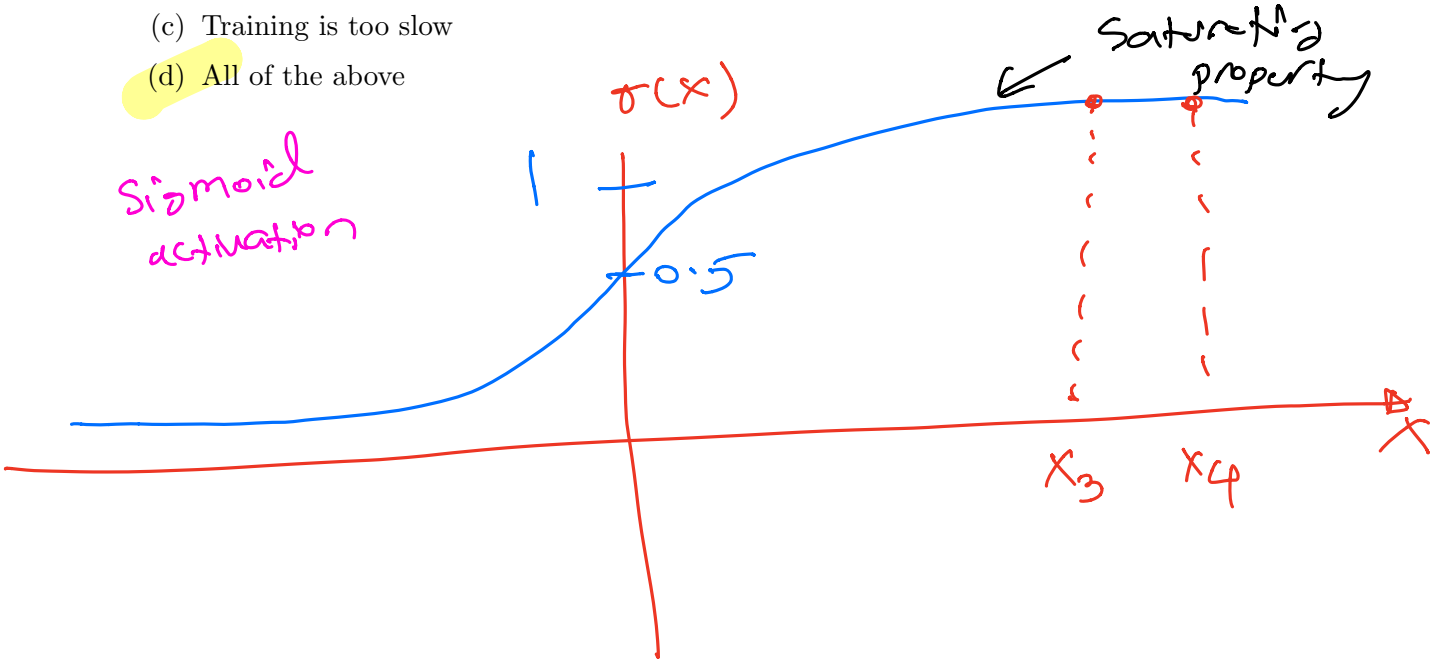
6. Which of the following neural network training challenge can be solved using batch normalization?

   (a) Overfitting

   (b) Restrict activations to become too high or low

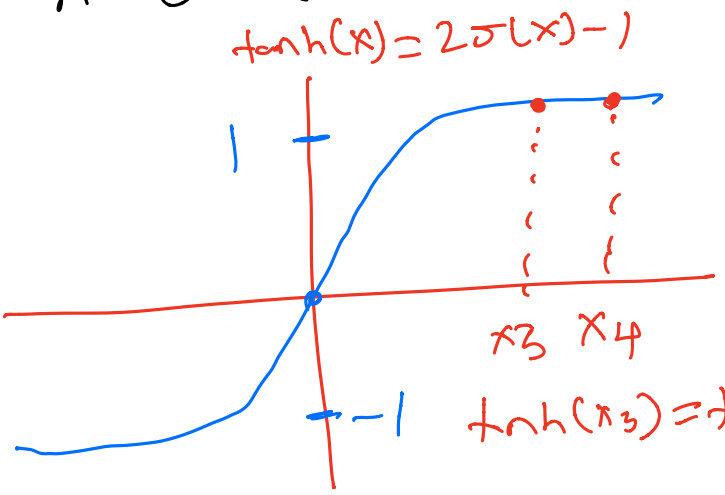
   (c) Training is too slow

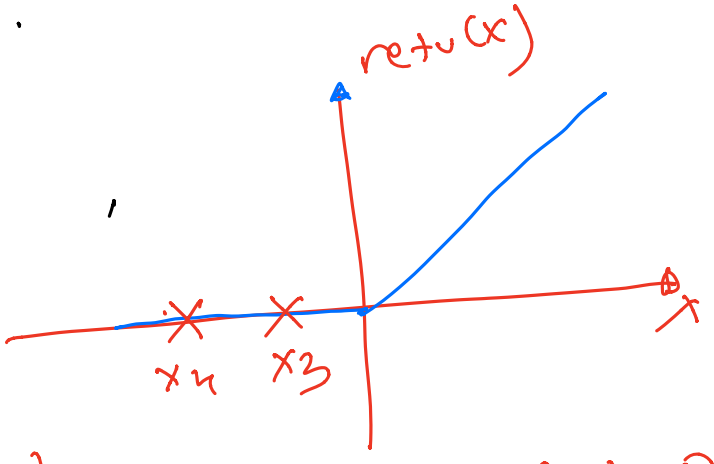
   (d) All of the above

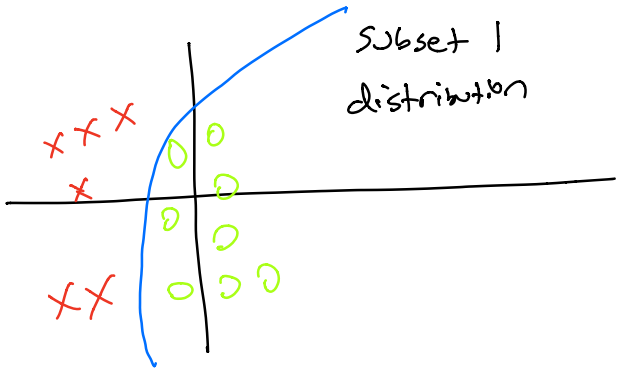


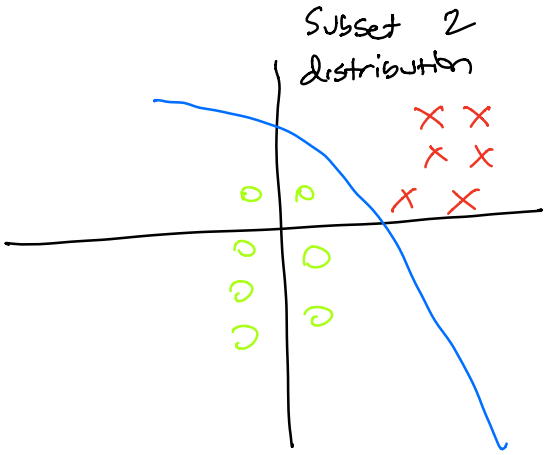
$\sigma(x_3) = \sigma(x_4) = 1$

Due to saturating property you can't classify because all high scores will have a value of 1.









- Subset 1 and Subset 2 lie in different regions of the feature space

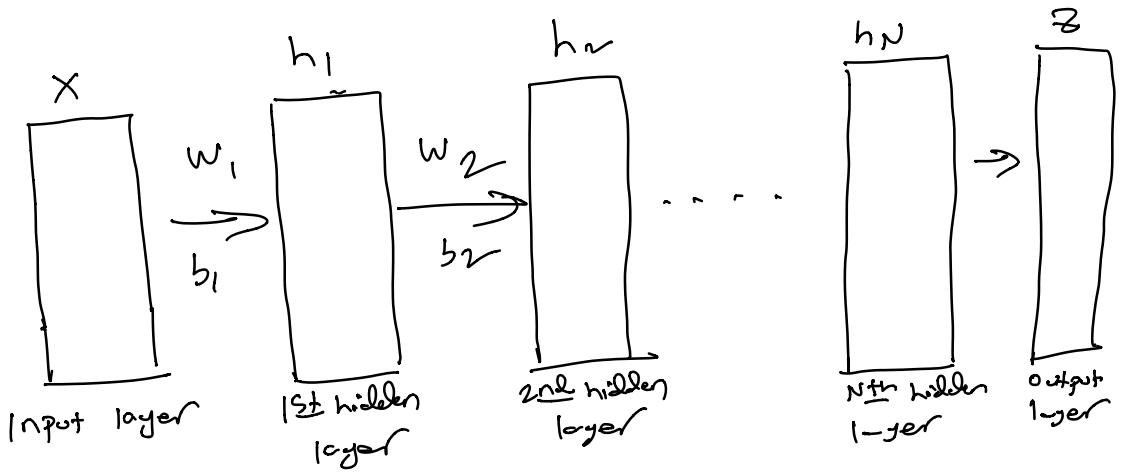
- This diff. in distribution is called /covariate shift/.

What happens if I train my classifier on Subset 1 and then test it on Subset 2 ?

Since they have very different distributions so we don't expect the model to generalize well.

Then what is the solution to the above problem?

If my minibatches have images that are randomly sampled from the entire dataset then the training set distribution is representative of both subset 1 and subset 2. Hence there will be a negligible covariate shift.



- By randomly selecting mini batches I can ensure that input layer has negligible covariate shift

$$h_1 = f(W_1 X + b_1)$$

$$h_2 = f(W_2 h_1 + b_2)$$ ← let's consider this

$$\vdots$$

$$h_N = f(W_N h_{N-1} + b_N)$$

Let's consider the second hidden layer: h_2

$$h_2 = f(W_2 h_1 + b_2)$$

Input to h_2 : h_1

Now, $h_1 = f(W_1 X + b_1)$ → update using GD

so every time I update my weight W_1 using gradient descent,

$$W_1^{(K+1)} = W_1^{(K)} - \varepsilon \frac{\partial \mathcal{L}}{\partial W_1}\Big|_K$$

the distribution of h_1 changes

Hence the input distribution to h_2 changes as training progresses.

This change in input distribution is called / Internal Covariate Shift /

What are some consequences of internal covariate shift?

- You can run into the issue of large/exploding gradients because your activations can become large. Therefore you can only pick a very small learning rate
- Slow convergence
- Sensitive to parameter initialization

The above consequences are observed during experiments and doesn't have theoretical guarantees.

BN addresses the problem of ICS by normalizing the input to each layer.

Let $x \in B$ denote an input to a BN layer, then it is transformed as follows

$$BN(x) = \gamma \odot \frac{x - \mu_B}{\sigma_B} + \beta$$

$$\mu_B = \frac{1}{|B|} \sum_{x \in B} x \quad \leftarrow \text{minibatch sample mean}$$

$$\sigma_B^2 = \frac{1}{|B|} \sum_{x \in B} (x - \mu_B)^2 + \varepsilon \quad \leftarrow \text{minibatch sample variance}$$

γ: Scaling Parameter
β: Shifting Parameter
} → Learnable Parameters

If $\gamma = 1$, $\beta = 0$ then BN layer normalizes the activations to gaussian statistics.

However constraining $\gamma = 1$ and $\beta = 0$ might reduce the capacity of my model.

Therefore, we introduce γ and β as learnable parameters such that we can learn them during training to best fit my data.

In HW4, check the values of γ and β.

Finally,

$$h_i = f(BN(W_i h_{i-1} + b_i))$$

Non linearity · Batch norm · Affine

$\leftarrow$ with BN

$$h_i = f(W_i h_{i-1} + b_i)$$

Non linearity · Affine

$\leftarrow$ without BN.