Published in Towards Data Science

Ketan Doshi Follow

May 18, 2021 · 9 min read · Listen

HANDS-ON TUTORIALS, INTUITIVE DEEP LEARNING SERIES

Batch Norm Explained Visually — How it works, and why neural networks need it

A Gentle Guide to an all-important Deep Learning layer, in Plain English

Photo by Reuben Teo on Unsplash

Batch Norm is a neural network layer that is now commonly used in many architectures. It often gets added as part of a Linear or Convolutional block and helps to stabilize the network during training.

In this article, we will explore what Batch Norm is, why we need it and how it works.

You might also enjoy reading my other article on Batch Norm which explains *why* Batch Norm works so well.

Batch Norm Explained Visually — Why does it work

A Gentle Guide to the reasons for the Batch Norm layer's success in making training converge faster, in Plain English

towardsdatascience.com

And if you're interested in Neural Network architectures in general, I have some other articles you might like.

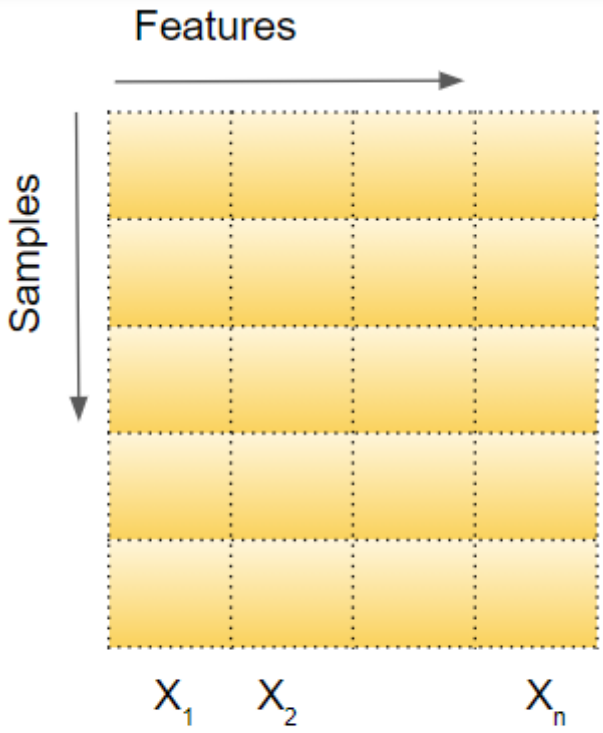
1. Optimizer Algorithms *(Fundamental techniques used by gradient descent optimizers like SGD, Momentum, RMSProp, Adam, and others)*
2. Differential and Adaptive Learning Rates (*How Optimizers and Schedulers can be used to boost model training and tune hyperparameters*)

But before we talk about Batch Normalization itself, let's start with some background about Normalization.

Normalizing Input Data

When inputting data to a deep learning model, it is standard practice to normalize the data to zero mean and unit variance. What does this mean and why do we do this?

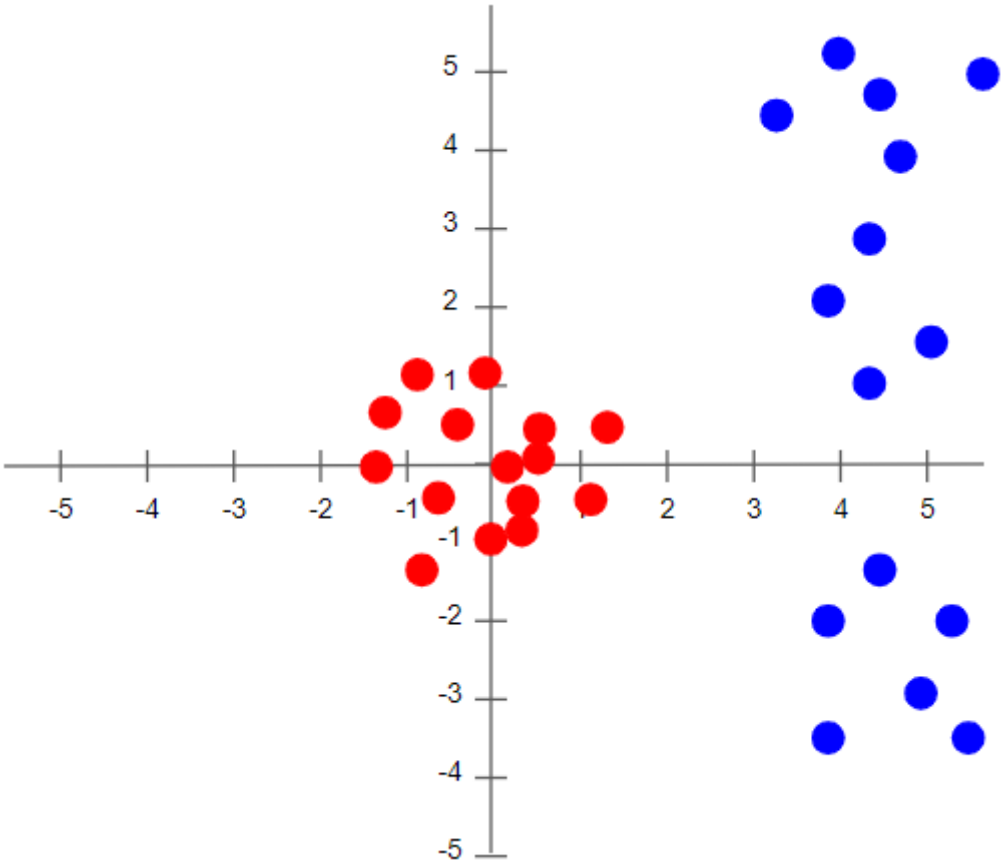
Let's say the input data consists of several features x1, x2,…xn. Each feature might have a different range of values. For instance, values for feature x1 might range from 1 through 5, while values for feature x2 might range from 1000 to 99999.



$$X_i = \frac{X_i - Mean_i}{StdDev_i}$$

How we normalize (Image by Author)

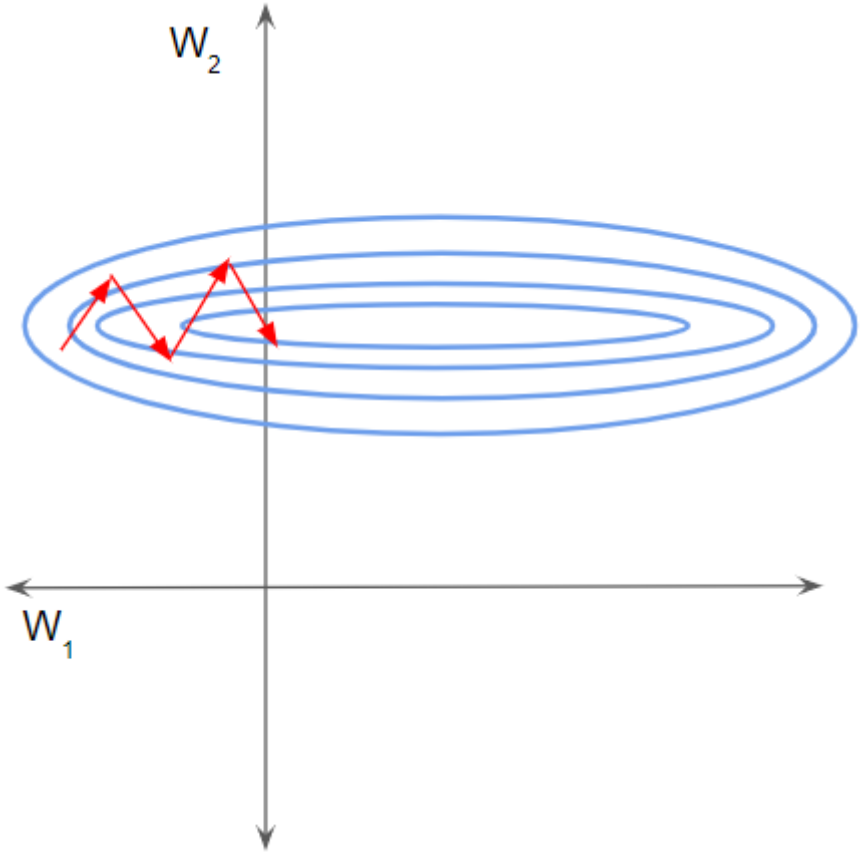
In the picture below, we can see the effect of normalizing data. The original values (in blue) are now centered around zero (in red). This ensures that all the feature values are now on the same scale.



What normalized data looks like (Image by Author)

To understand what happens without normalization, let's look at an example with just two features that are on drastically different scales. Since the network output is a linear combination of each feature vector, this means that the network learns weights for each feature that are also on different scales. Otherwise, the large feature will simply drown out the small feature.

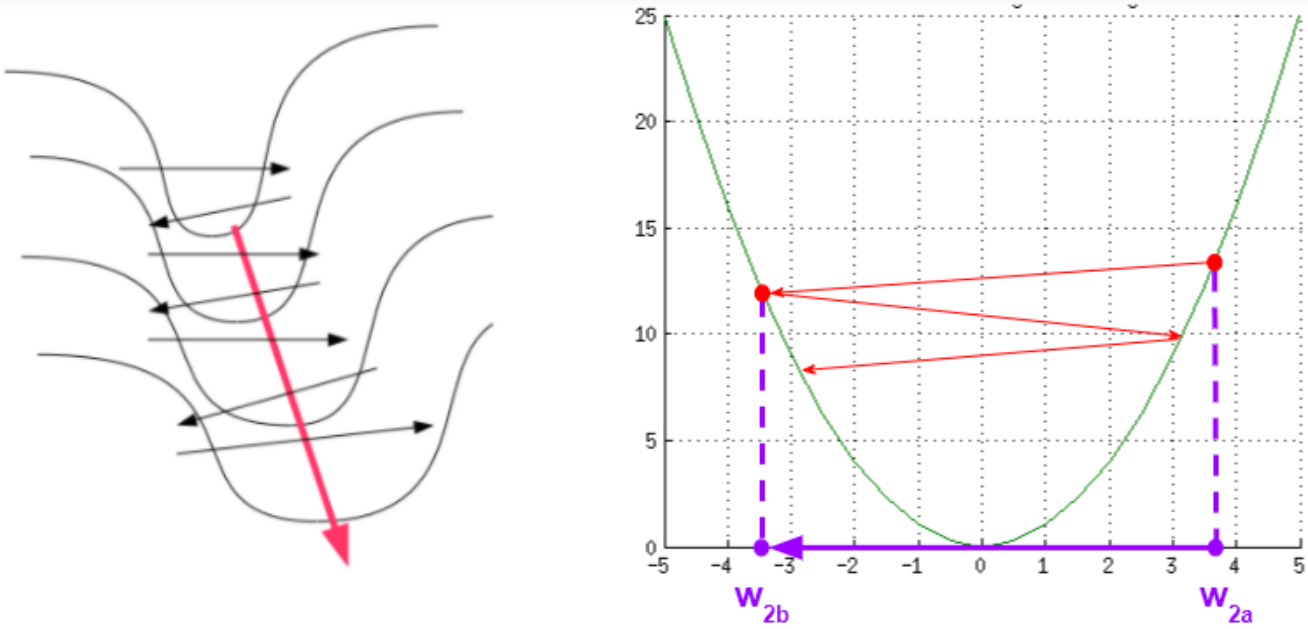
Then during gradient descent, in order to "move the needle" for the Loss, the network would have to make a large update to one weight compared to the other weight. This can cause the gradient descent trajectory to oscillate back and forth along one dimension, thus taking more steps to reach the minimum.



Features on different scales take longer to reach the minimum (Image by Author)

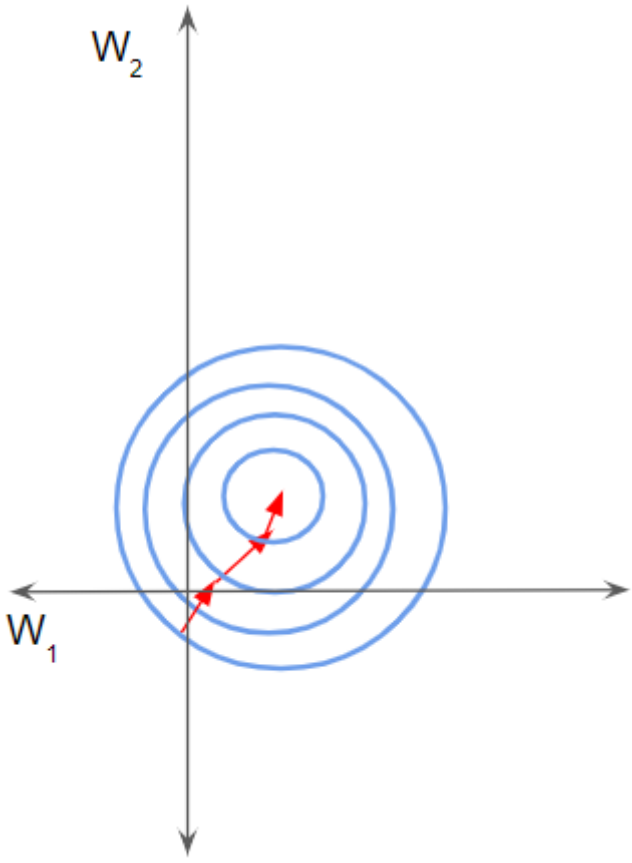
In this case, the loss landscape looks like a narrow ravine. We can decompose the gradient along the two dimensions. It is steep along one dimension and much more gentle along the other dimension.

We end up making a larger update to one weight due to its large gradient. This causes the gradient descent to bounce to the other side of the slope. On the other hand, the smaller gradient along the second direction results in us making smaller weight updates and thus taking smaller steps. This uneven trajectory takes longer for the network to converge.



A narrow valley causes gradient descent to bounce from one slope to the other (Image by Author)

Instead, if the features are on the same scale, the loss landscape is more uniform like a bowl. Gradient descent can then proceed smoothly down to the minimum.



Normalized data helps the network converge faster (Image by Author)

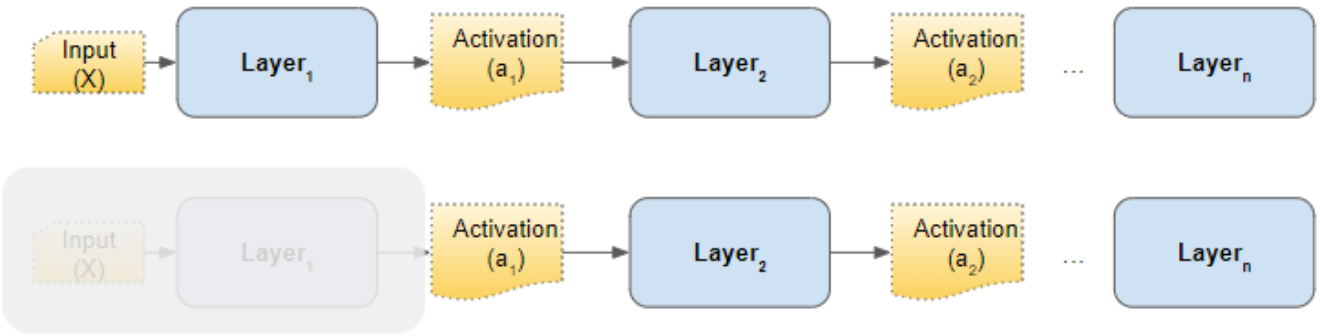
evolved to tackle these challenges.

The need for Batch Norm

Now that we understand what Normalization is, the reason for needing Batch Normalization starts to become clear.

Consider any of the hidden layers of a network. The activations from the previous layer are simply the inputs to this layer. For instance, from the perspective of Layer 2 in the picture below, if we "blank out" all the previous layers, the activations coming from Layer 1 are no different from the original inputs.

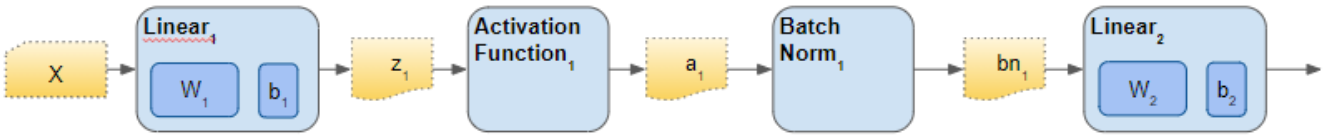
The same logic that requires us to normalize the input for the first layer will also apply to each of these hidden layers.



The inputs of each hidden layer are the a ious layer, and must also be normalized

In other words, if we are able to somehow normalize the activations from each previous layer then the gradient descent will converge better during training. This is precisely what the Batch Norm layer does for us.

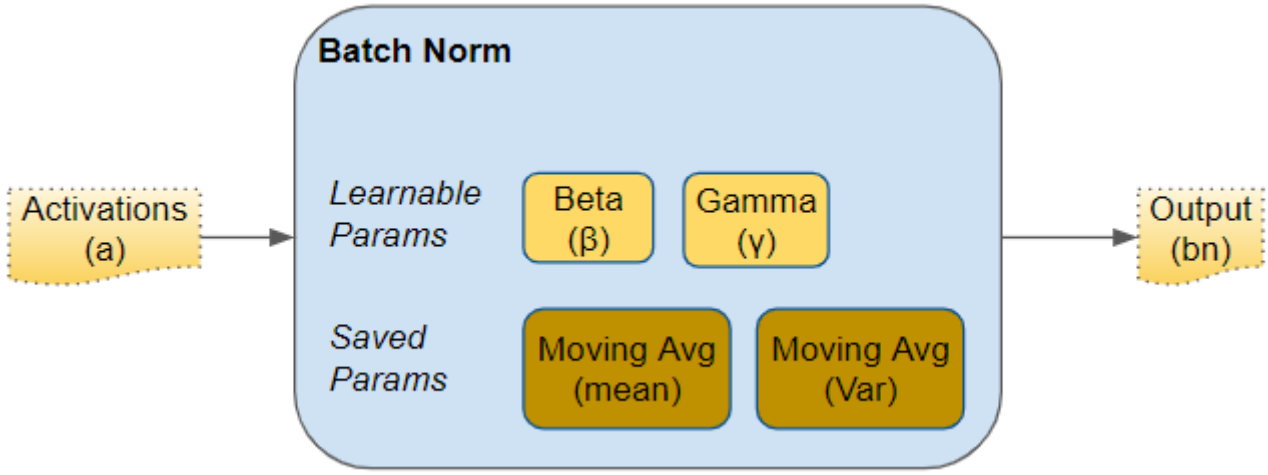
normalize them before passing them on as the input of the next hidden layer.



The Batch Norm layer normalizes activations from Layer 1 before they reach layer 2 (Image by Author)

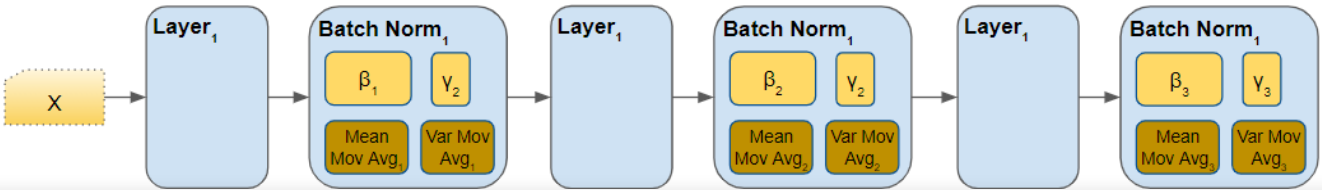
Just like the parameters (eg. weights, bias) of any network layer, a Batch Norm layer also has parameters of its own:

- Two learnable parameters called beta and gamma.
- Two non-learnable parameters (Mean Moving Average and Variance Moving Average) are saved as part of the 'state' of the Batch Norm layer.

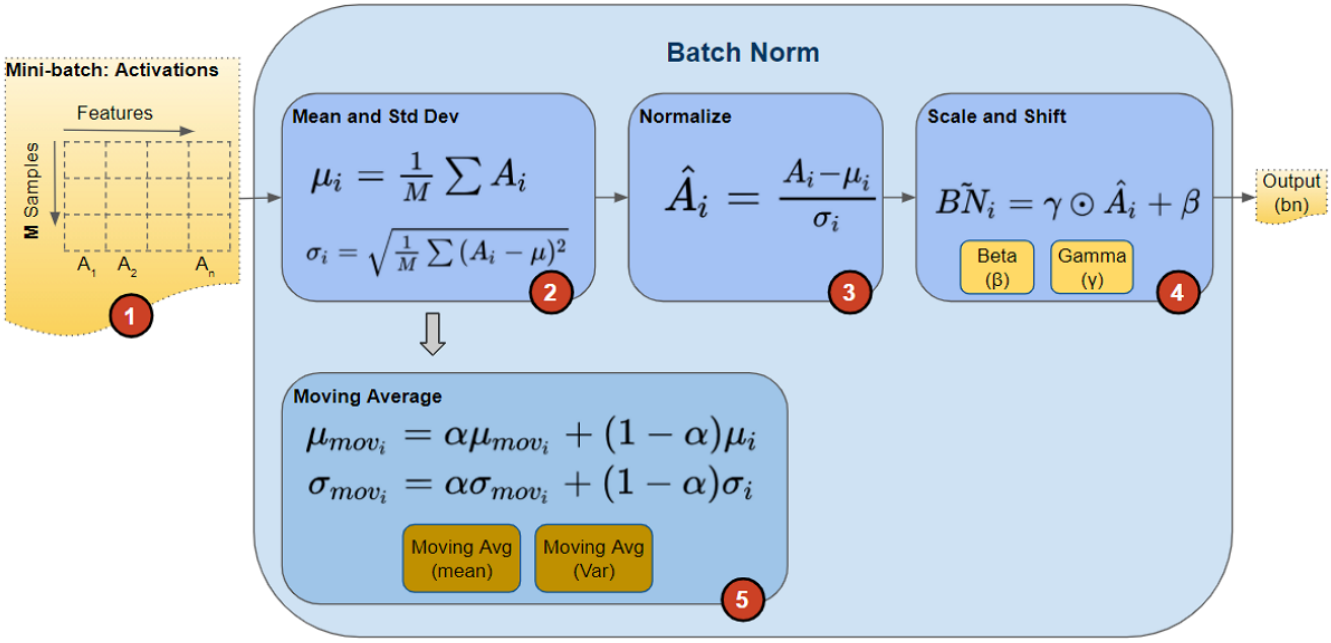


Parameters of a Batch Norm layer (Image by Author)

These parameters are per Batch Norm layer. So if we have, say, three hidden layers and three Batch Norm layers in the network, we would have three learnable beta and gamma parameters for the three layers. Similarly for the Moving Average parameters.



During training, we feed the network one mini-batch of data at a time. During the forward pass, each layer of the network processes that mini-batch of data. The Batch Norm layer processes its data as follows:



Calculations performed by Batch Norm layer (Image by Author)

1. Activations

The activations from the previous layer are passed as input to the Batch Norm. There is one activation vector for each feature in the data.

2. Calculate Mean and Variance

For each activation vector separately, calculate the mean and variance of all the values in the mini-batch.

3. Normalize

Calculate the normalized values for each activation feature vector using the corresponding mean and variance. These normalized values now have zero mean and unit variance.

4. Scale and Shift

This step is the huge innovation introduced by Batch Norm that gives it its power. Unlike the input layer, which requires all normalized values to have zero mean and unit

gamma, and adding to it a factor, beta. Note that this is an element-wise multiply, not a matrix multiply.

What makes this innovation ingenious is that these factors are not hyperparameters (ie. constants provided by the model designer) but are trainable parameters that are learned by the network. In other words, each Batch Norm layer is able to optimally find the best factors for itself, and can thus shift and scale the normalized values to get the best predictions.

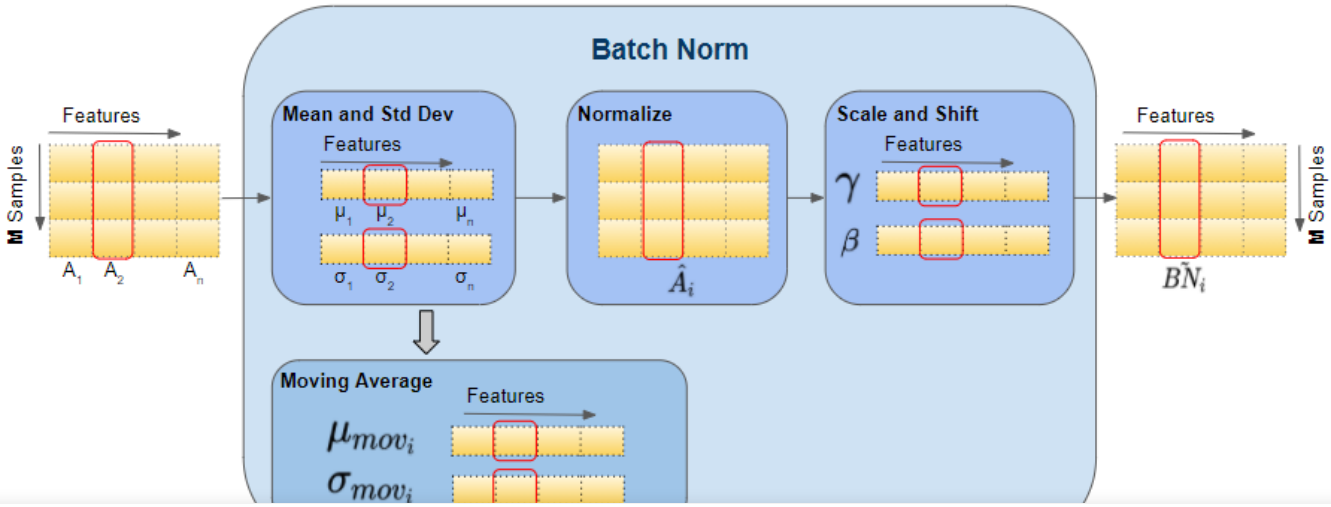
5. Moving Average

In addition, Batch Norm also keeps a running count of the Exponential Moving Average (EMA) of the mean and variance. During training, it simply calculates this EMA but does not do anything with it. At the end of training, it simply saves this value as part of the layer's state, for use during the Inference phase.

We will return to this point a little later when we talk about Inference. The Moving Average calculation uses a scalar 'momentum' denoted by alpha below. This is a hyperparameter that is used only for Batch Norm moving averages and should not be confused with the momentum that is used in the Optimizer.

Vector Shapes

Below, we can see the shapes of these vectors. The values that are involved in computing the vectors for a particular feature are also highlighted in red. However, remember that all feature vectors are computed in a single matrix operation.

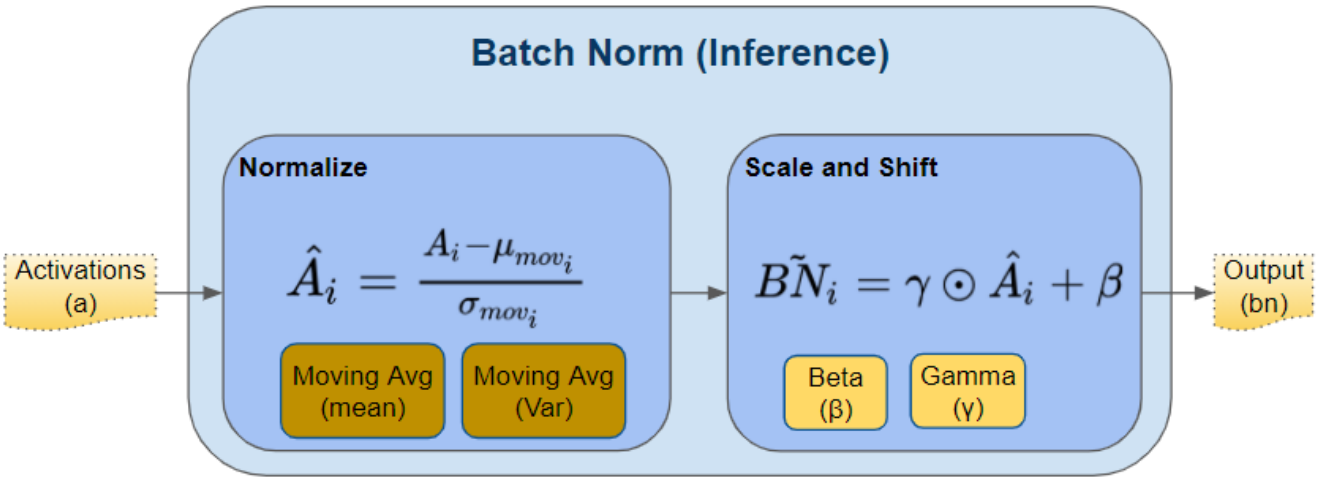


After the forward pass, we do the backward pass as normal. Gradients are calculated and updates are done for all layer weights, as well as for all beta and gamma parameters in the Batch Norm layers.

Batch Norm during Inference

As we discussed above, during Training, Batch Norm starts by calculating the mean and variance for a mini-batch. However, during Inference, we have a single sample, not a mini-batch. How do we obtain the mean and variance in that case?

Here is where the two Moving Average parameters come in — the ones that we calculated during training and saved with the model. We use those saved mean and variance values for the Batch Norm during Inference.

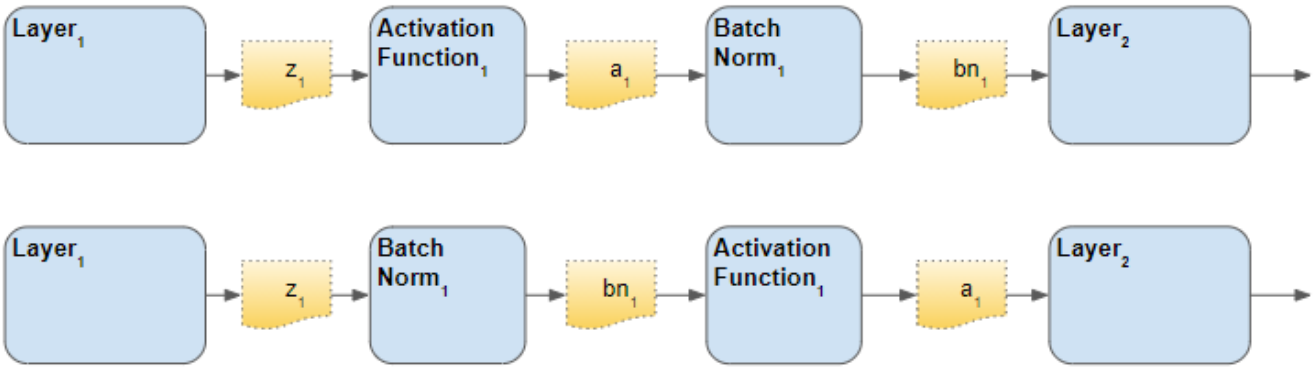


Batch Norm calculation during Inference (Image by Author)

Ideally, during training, we could have calculated and saved the mean and variance for the full data. But that would be very expensive as we would have to keep values for the full dataset in memory during training. Instead, the Moving Average acts as a good proxy for the mean and variance of the data. It is much more efficient because the calculation is incremental — we have to remember only the most recent Moving Average.

Order of placement of Batch Norm layer

There are two opinions for where the Batch Norm layer should be placed in the



Batch Norm can be used before or after activation (Image by Author)

Conclusion

Batch Norm is a very useful layer that you will end up using often in your network architecture. Hopefully, this gives you a good understanding of *how* Batch Norm works.

It is also useful to understand *why* Batch Norm helps in network training, which I will cover in detail in another article.

And finally, if you liked this article, you might also enjoy my other series on Transformers, Audio Deep Learning, and Geolocation Machine Learning.

Transformers Explained Visually (Part 1): Overview of Functionality

A Gentle Guide to Transformers for NLP, and why they are better than RNNs, in Plain English. How Attention helps...

towardsdatascience.com

Audio Deep Learning Made Simple (Part 1): State-of-the-Art Techniques

A Gentle Guide to the world of disruptive deep learning audio applications and architectures. And why we all need to...

towardsdatascience.com

Let's keep learning!

Published in Towards Data Science

Ketan Doshi Follow

May 26, 2021 · 9 min read · Listen

HANDS-ON TUTORIALS, INTUITIVE DEEP LEARNING SERIES

Batch Norm Explained Visually — Why does it work?

A Gentle Guide to the reasons for the Batch Norm layer's success in making training converge faster, in Plain English

Photo by AbsolutVision on Unsplash

such as Inception and Resnet rely on it to create deeper networks that can be trained faster.

In this article, we will explore *why* Batch Norm works and why it requires fewer training epochs when training a model.

You might also enjoy reading my other article on Batch Norm which explains, in simple language, what Batch Norm is and walks through, step by step, how it operates under the hood.

Batch Norm Explained Visually — How it works, and why neural networks need it

A Gentle Guide to an all-important Deep Learning layer, in Plain English

towardsdatascience.com

And if you're interested in Neural Network architectures in general, I have some other articles you might like.

1. Optimizer Algorithms *(Fundamental techniques used by gradient descent optimizers like SGD, Momentum, RMSProp, Adam, and others)*
2. Image Captions Architecture *(Multi-modal CNN and RNN architectures with Image Feature Encoders, Sequence Decoders, and Attention)*

Why does Batch Norm work?

There is no dispute that Batch Norm works wonderfully well and provides substantial measurable benefits to deep learning architecture design and training. However, curiously, there is still no universally agreed answer about what gives it its amazing powers.

To be sure, many theories have been proposed. But over the years, there is disagreement about which of those theories is the right one.

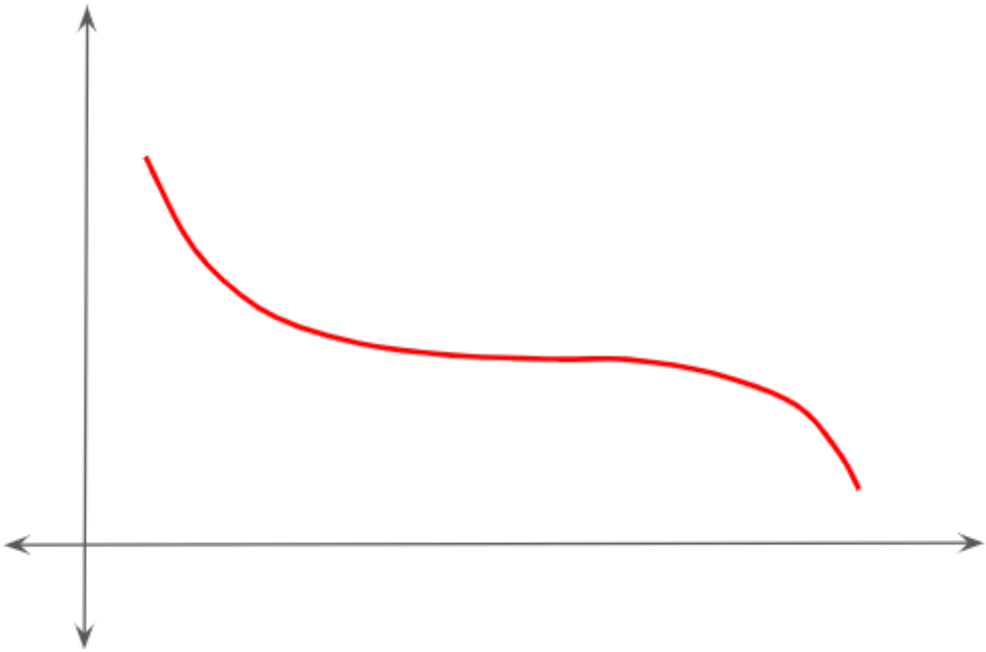
The first explanation by the original inventors for why Batch Norm worked, was based

Smoothening of the Loss and Gradient curves. These are the two most well-known hypotheses, so let's go over them below.

Theory 1 — Internal Covariate Shift

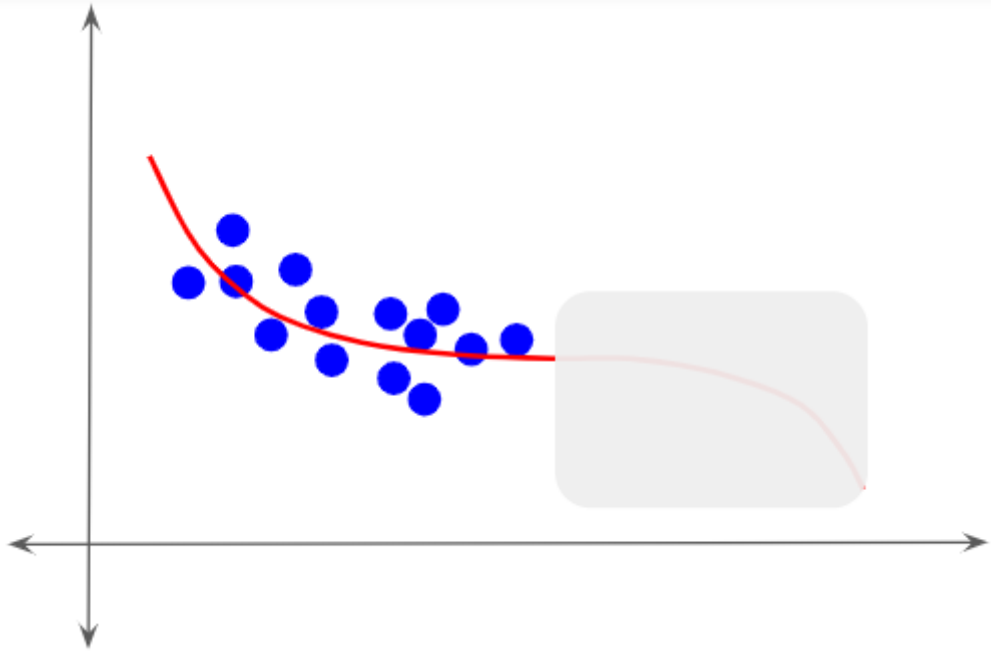
If you're like me, I'm sure you find this terminology quite intimidating! 😊 What does it mean, in simple language?

Let's say that we want to train a model and the ideal target output function (although we don't know it ahead of time) that the model needs to learn is as below.



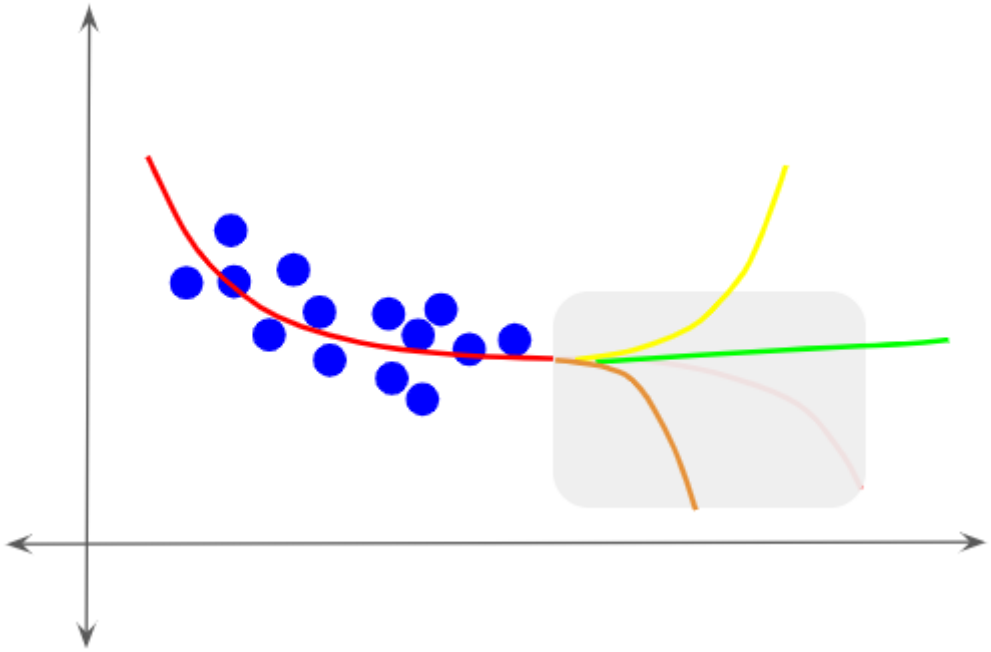
Target function (Image by Author)

Somehow, let's say that the training data values that we input to the model cover only a part of the range of output values. The model is, therefore, able to learn only a subset of the target function.



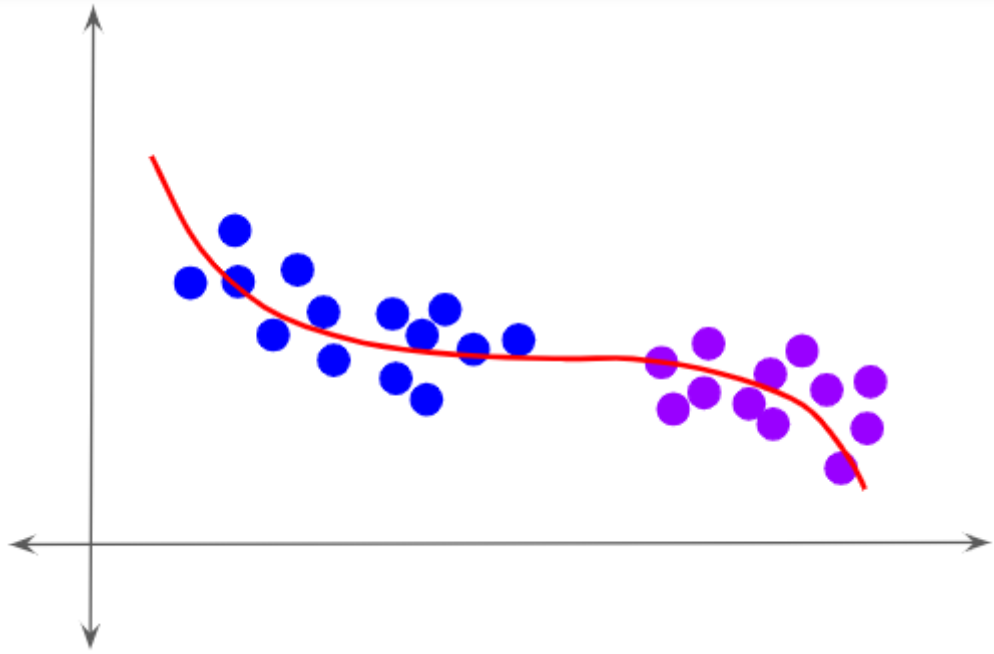
Training data distribution (Image by Author)

The model has no idea about the rest of the target curve. It could be anything.



Rest of the target curve (Image by Author)

Suppose that we now feed the model some different testing data as below. This has a very different distribution from the data that the model was initially trained with. The model is not able to generalize its predictions for this new data.



Test data has di… …age by Author)

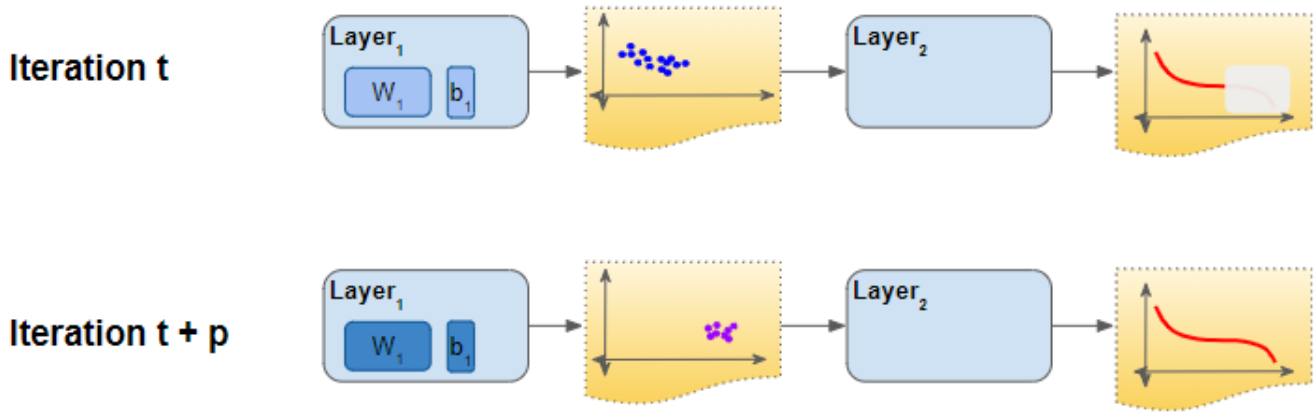
This is the problem of Covariate Shift — the model is fed data with a very different distribution than what it was previously trained with — *even though that new data still conforms to the same target function*.

For the model to figure out how to adapt to this new data, it has to re-learn some of its target output function. This slows down the training process. Had we provided the model with a representative distribution that covered the full range of values from the beginning, it would have been able to learn the target output sooner.

Now that we understand what "Covariate Shift" is, let's see how it affects network training.

During training, each layer of the network learns an output function to fit its input. Let's say that during one iteration, Layer 'k' receives a mini-batch of activations from the previous layer. It then adjusts its output activations by updating its weights based on that input.

However, in each iteration, the previous layer 'k-1' is also doing the same thing. It adjusts its output activations, effectively changing its distribution.



How Covariate Shift affects Training (Image by Author)

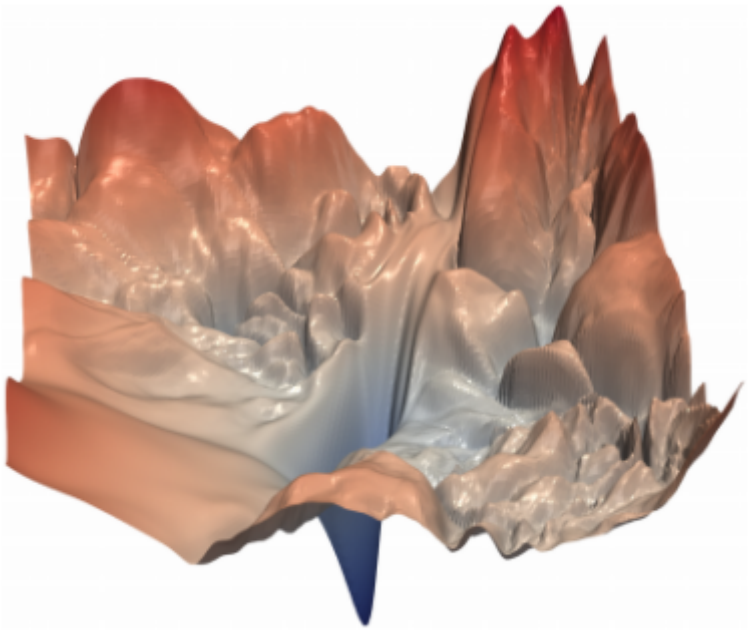
That is also the input for layer 'k'. In other words, that layer receives input data that has a different distribution than before. It is now forced to learn to fit to this new input. As we can see, each layer ends up trying to learn from a constantly shifting input, thus taking longer to converge and slowing down the training.

Therefore the proposed hypothesis was that Batch Norm helps to stabilize these shifting distributions from one iteration to the next, and thus speeds up training.

Theory 2 — Loss and Gradient Smoothening

The MIT paper published results that challenge the claim that addressing Covariate Shift is responsible for Batch Norm's performance, and puts forward a different explanation.

In a typical neural network, the "loss landscape" is not a smooth convex surface. It is very bumpy with sharp cliffs and flat surfaces. This creates a challenge for gradient descent — because it could suddenly encounter an obstacle in what it thought was a promising direction to follow. To compensate for this, the learning rate is kept low so that we take only small steps in any direction.



A neural network loss landscape (Source, by permission of Hao Li)

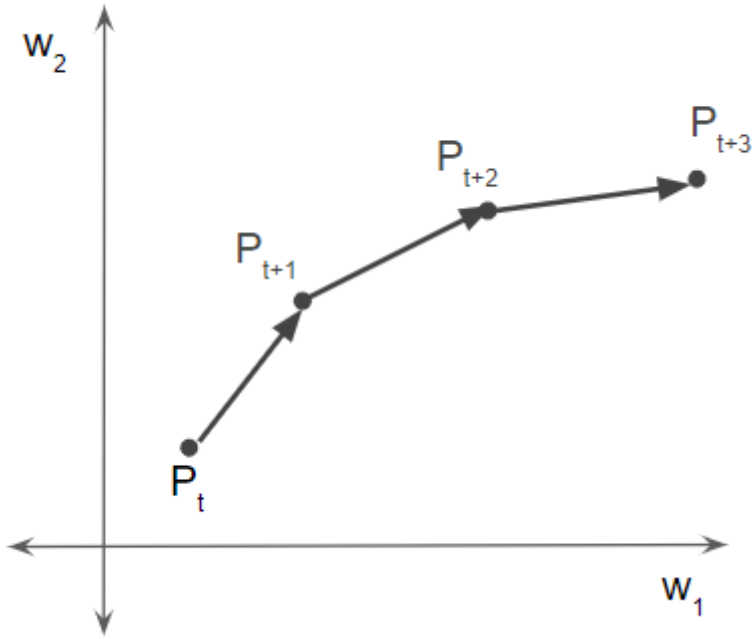
If you would like to read more about this, please see my article on neural network Optimizers that explains this in more detail, and how different Optimizer algorithms have evolved to tackle these challenges.

The paper proposes that what Batch Norm does is to smoothen the loss landscape substantially by changing the distribution of the network's weights. This means that gradient descent can confidently take a step in a direction knowing that it will not find abrupt disruptions along the way. It can thus take larger steps by using a bigger learning rate.

To investigate this theory, the paper conducted an experiment to analyze the loss landscape of a model during training. We'll try and visualize this with a simple example.

Let's say that we have a simple network with two weight parameters (w1 and w2). The values of these weights can be shown on a 2D surface, with one axis for each weight. Each combination of weight values corresponds to a point on this 2D plane.

As the weights change during training, we move to another point on this surface. Hence one can plot the trajectory of the weights over the training iterations.



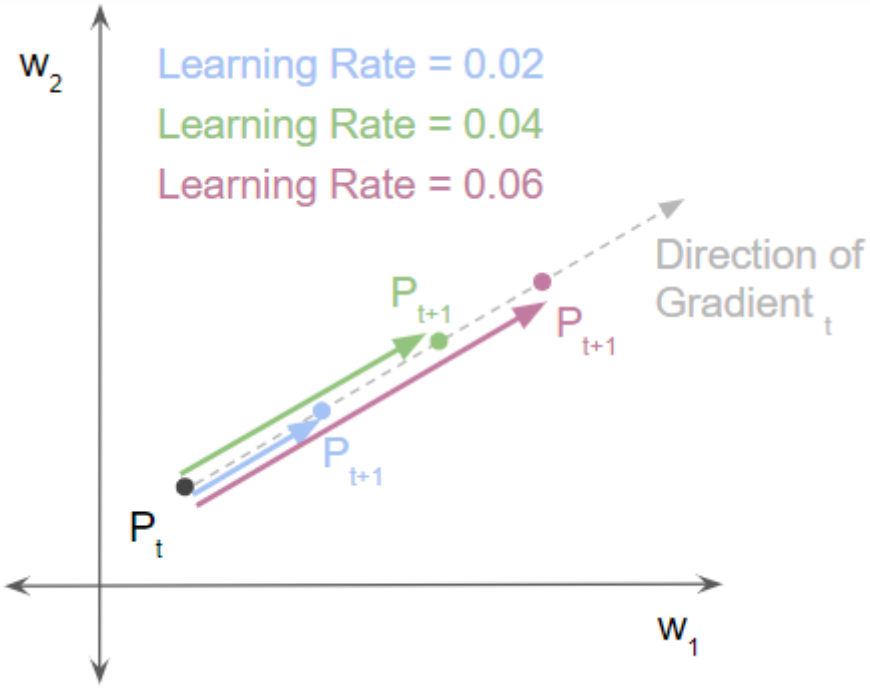
(Image by Author)

The goal of the experiment was to examine the loss landscape, by measuring what the Loss and Gradient would look like at different points if we kept moving in the same direction. They measured this with and without Batch Norm, to see what effect Batch Norm had.

Let's say that at some iteration 't' during training it is at point P(t). It evaluates the loss and the gradient at Pt. Then, starting from that point, it takes a single step forward with some learning rate to reach the next point P(t+1). Then it does a rewind back to P(t) and repeats the step with a higher learning rate.

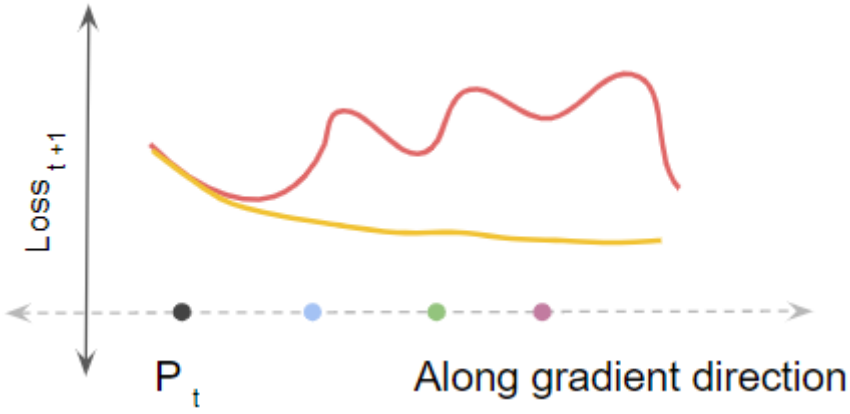
In other words, it tried three different alternatives by taking three different-sized steps (blue, green, and pink arrows) along the gradient direction, using three different learning rates. That brought us to three different next points for P(t+1). Then, at each of those P(t+1) points, it measured the new loss and gradient.

After this, it repeated all three steps for the same network, but with Batch Norm included.



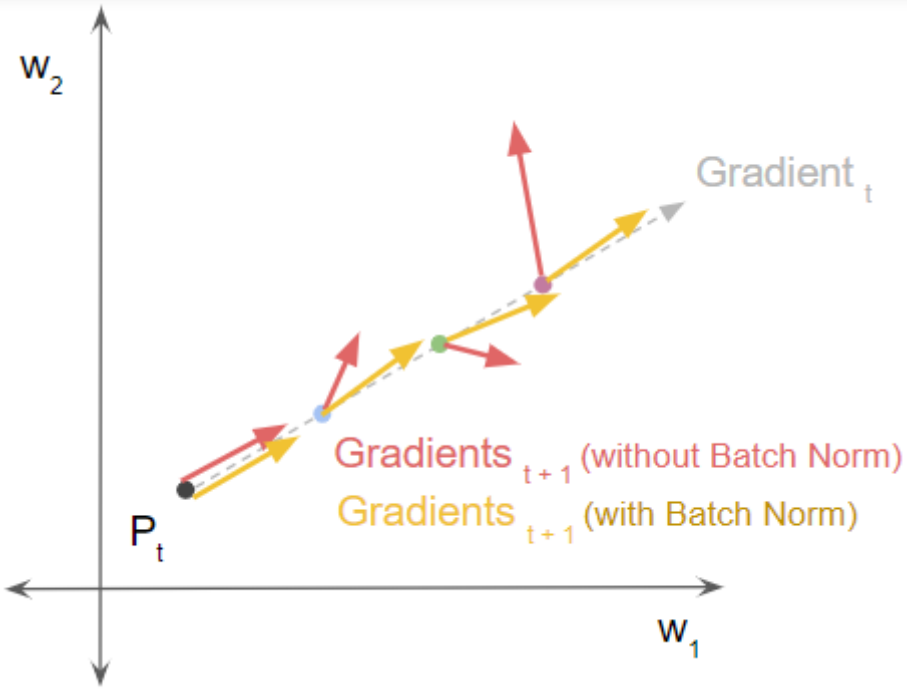
(Image by Author)

Now we can plot the loss at each of those P(t+1) points (blue, green, and pink) just in that single direction. The bumpy red curve shows the loss without Batch Norm and the smooth declining yellow curve shows the loss with Batch Norm.



Loss with Batch declines smoothly (Image by Author)

Similarly, we can plot the size and direction of the gradient at each of those points. The red arrows show the gradient fluctuating wildly in size and direction without Batch Norm. The yellow arrows show the gradient remaining steady in size and direction with Batch Norm.



Gradients with Batch Norm are smoother (Image by Author)

This experiment shows us that Batch Norm significantly smoothens the loss landscape. How does that help our training?

The ideal scenario is that at the next point P(t+1), the gradient also lies in the same direction. This means that we can continue moving in the same direction. This allows the training to proceed smoothly and quickly find the minimum.

On the other hand, if the best gradient direction at P(t+1) took us in a different direction, we would end up following a zig-zag route with wasted effort. That would require more training iterations to converge.

Although this paper's findings have not been challenged so far, it isn't clear whether they've been fully accepted as conclusive proof to close this debate.

Regardless of which theory is correct, what we do know for sure is that Batch Norm provides several advantages.

Advantages of Batch Norm

The huge benefit that Batch Norm provides is to allow the model to converge faster and speed up training. It makes the training less sensitive to how the weights are initialized

descent. Batch Norm helps to reduce the effect of these outliers.

Batch Norm also reduces the dependence of gradients on the initial weight values. Since weights are initialized randomly, outlier weight values in the early phases of training can distort gradients. Thus it takes longer for the network to converge. Batch Norm helps to dampen the effects of these outliers.

When is Batch Norm not applicable?

Batch Norm doesn't work well with smaller batch sizes. That results in too much noise in the mean and variance of each mini-batch.

Batch Norm is not used with recurrent networks. Activations after each timestep have different distributions, making it impractical to apply Batch Norm to it.

Conclusion

Even though we aren't sure about the correct explanation, it is fascinating to explore these different theories as it gives us some insight into the inner workings of neural networks.

Regardless, Batch Norm is a layer that we should definitely consider when designing our architecture.

And finally, if you liked this article, you might also enjoy my other series on Transformers, Audio Deep Learning, and Geolocation Machine Learning.

Transformers Explained Visually (Part 1): Overview of Functionality

A Gentle Guide to Transformers for NLP, and why they are better than RNNs, in Plain English. How Attention helps...

towardsdatascience.com

Audio Deep Learning Made Simple (Part 1): State-of-the-Art Techniques

A Gentle Guide to the world of disruptive deep learning audio

Leveraging Geolocation Data for Machine Learning: Essential Techniques

A Gentle Guide to Feature Engineering and Visualization with Geospatial data, in Plain English

towardsdatascience.com

Let's keep learning!

Lei Mao

Artificial Intelligence
Machine Learning
Computer Science

Santa Clara, California

POSTS	CATEGORIES	TAGS
431	8	262

CATALOGUE

1 Introduction
2 Motivation of Batch Normalization
3 Mathematical Definition
4 Caveats
5 References

Batch Normalization Explained

09-07-2018 05-31-2019 BLOG 6 MINUTES READ (ABOUT 961 WORDS) 1442 VISITS

Introduction

Recently I was working on a collaborative deep learning project trying to reproduce a model from the publication, but I found the model was overfit significantly. My dear colleague examined my code and pointed out that there might be some problems in my Tensorflow batch normalization implementation. After checking the possible correct implementation, I realized that probably I did not fully understand batch normalization. In this blog post, I am going to review batch normalization again on its mathematical definition and intuitions.

Motivation of Batch Normalization

I am not going to explain why batch normalization works well in real practice, since Andrew Ng has a very good video explaining that.

Mathematical Definition

Training Phase

Given inputs x over a minibatch of size m, $B = \{x_1, x_2, \dots, x_m\}$, by applying transformation of your inputs using some learned parameters γ and β, the outputs could be expressed as $B' = \{y_1, y_2, \dots, y_m\}$, where $y_i = \text{BN}_{\gamma,\beta}(x_i)$.

More concretely, we first calculate the mean and the variance of the samples from the minibatch.

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i$$
$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$$

Then we normalize the samples to zero means and unit variance. ϵ is for numerical stability in case the denominator becomes zero by chance.

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

Finally, a little bit surprising, there is a scaling and shifting step. γ and β are learnable parameters.

$$y_i = \gamma\hat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i)$$

Test Phase

During the test phase, specifically when you only have one test sample, doing batch normalization as the one in the training phase does not make sense, because your outputs at each layer of the network will be exactly zero. To overcome this, people invented "running mean" and "running variance", which are updated in real-time during training.

More concretely, at training time step t, the running mean $\mu'_B[t]$ and running variance $\sigma'^2_B[t]$ are calculated as follows:

$$\mu'_B[t] = \mu'_B[t] \times \text{momentum} + \mu_B[t] \times (1 - \text{momentum})$$
$$\sigma'^2_B[t] = \sigma'^2_B[t] \times \text{momentum} + \sigma^2_B[t] \times (1 - \text{momentum})$$

Here momentum is sometimes also called decay.

The "running mean" and "running variance" were computed during training and would be used during inference.

Caveats

Value of Momentum

Surprisingly, momentum is a very important parameter for model validation performance. In TensorFlow, it suggests how to set momentum correctly.

decay: Decay for the moving average. Reasonable values for decay are close to 1.0, typically in the multiple-nines range: 0.999, 0.99, 0.9, etc. Lower decay value (recommend trying decay=0.9) if the model experiences reasonably good training performance but poor validation and/or test performance. Try zero_debias_moving_mean=True for improved stability.

Intuitively, when momentum = 0, the running means and variances are always the means and variance of the last minibatch. This running means and variance could be highly biased and thus the training performance and validation performance differ significantly. When momentum = 1.0, the running means and variances are always the means and variance of the first minibatch, which could also be highly biased. So the momentum value should not be extremely close to 0 or 1.0. In addition, because we want to "average" over as many samples as possible and the samples in the past mini-batches are important which should be given more weights, the momentum value should be a large number close to 1.0. The momentum could be thought of as a weight factor for the past and the present information! Therefore, taken together, a value of multiple nines are recommended for momentum.

Specify Training Mode and Test Mode

In TensorFlow, you will have to specify training mode and test mode when you are running your model in different stages.

There is also a very special setting in TensorFlow if you want to train a model that has a batch norm layer. Unfortunately, for some of my previous codes, I did not have these settings, which means that during the test stage, the samples were not probably normalized "correctly" as expected. Zero will be output from the batch norm layer if test samples are tested one by one, which will significantly affect the testing performance!

```
x_norm = tf.layers.batch_normalization(x, training=training)

# ...

update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
with tf.control_dependencies(update_ops):
    train_op = optimizer.minimize(loss)
```

Batch Normalization in Convolutional Neural Network

If batch normalization is working on the outputs from a convolution layer, the math has to be modified slightly since it does not make sense to calculate the mean and variance for every single pixel and do the normalization for every single pixel.

Assume the input tensor has shape $[m, H, W, C]$, for each channel $c \in \{1, 2, \cdots, C\}$

$$\mu_{B,c} = \frac{1}{mHW} \sum_{i=1}^{m} \sum_{j=1}^{H} \sum_{k=1}^{W} x_{i,j,k,c}$$

$$\sigma_{B,c}^2 = \frac{1}{mHW} \sum_{i=1}^{m} \sum_{j=1}^{H} \sum_{k=1}^{W} (x_{i,j,k,c} - \mu_{B,c})^2$$

$$\hat{x}_{i,j,k,c} = \frac{x_{i,j,k,c} - \mu_{B,c}}{\sqrt{\sigma_{B,c}^2 + \epsilon}}$$

Specifically for each channel, we have learnable parameters γ_c and β_c, such that

$$y_{i,:,:,c} = \gamma_c \hat{x}_{i,:,:,c} + \beta_c \equiv \text{BN}_{\gamma_c,\beta_c}(x_{i,:,:,c})$$

References

- Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift
- Batch Normalization TensorFlow Implementation

Author Lei Mao | Posted on 09-07-2018 | Updated on 05-31-2019 | Licensed under

DEEP LEARNING

Semantic Segmentation Using DeepLab V3 金门大桥徒步

Comments