Initialization and Normalization in Deep Neural Networks

Feb 28, 2017. Wei YANG

platero.yang (at) gmail.com

Table of Contents:

- Intro
- Initialization of the parameters
  - Naive initialization
  - Xavier initialization
  - ReLU: MSR initialization
- Normalization
  - Normalizing the inputs
  - Batch (Re)normalization: normalizing the input for each layer
  - Normalizing the weights
  - Normalizing in recurrent neural networks
- Outro

Introduction

In this note, we will focus on training neural networks efficiently by appropriate weight initialization and by the normalization techniques.

Initialization of the parameters

The initial value of the network parameters can affect the training process significantly. If the weights are initialized very large, then the nonlinear activation function (e.g., Sigmoid and Tanh) might saturate, which makes the gradients very small. If the weights are initialized very small, then the gradients would be small too. SGD methods with small gradients update the weight slowly, which slows down the training process. One might try to initialize all weights to zero. However, this is a common pitfall discussed in cs231n:
If every neuron in the network computes the same output, then they will all compute the same gradients during backpropagation and undergo the exact same parameter updates. In other words, there is no source of asymmetry between neurons if their weights are initialized to be the same.

Naive initialization

Intuitively, the weights are expected closed to zero. Hence we usually initialize the weights from a Gaussian distribution with zero mean and a small variance, e.g.,

$$W_{ij} \sim N(0, 0.01^2).$$

Although it is widely used in practice, there is no implicit evidence that why we set the standard variation as 0.01.

Xavier initialization

Our derivation mainly following Glorot and Bengio. The idea is to keep the input and output of a layer with the same variance and the zero mean. A building block of a conventional neural network consists of a linear layer and an elementwise activation function \(f(\cdot)\),

$$\begin{cases} y^l = w_l x^l + b^l, \\ x^{l+1} = f(y^l), \end{cases}$$

where \(x\) is the input and \(y\) is the output. \(W_l\) is the weight and \(b\) is the bias. \(l\) indexes the layer.

We drop the \(l\) for simplicity. We assume that the elements in \(W\) are independent and identically distributed (i.i.d.), and the elements in \(x\) are also i.i.d. And \(W\) and \(x\) are independent with each other. Suppose we use the sigmoid function as the nonlinear activation function, then \(x\) has zero mean \(E[(x_i)] = 0\). Since \(y = \sum_{i=1}^n W_i x_i\), we have
\[ \text{Var}(y) = \text{Var} \left( \sum_{i}^{n} w_i x_i \right) \]

\[ = \sum_{i}^{n} \text{Var}(w_i x_i) \]

\[ = \sum_{i}^{n} [E(w_i)]^2 \text{Var}(x_i) + E[(x_i)]^2 \text{Var}(w_i) + \text{Var}(x_i) \text{Var}(w_i) \]

\[ = \sum_{i}^{n} \text{Var}(x_i) \text{Var}(w_i) \]

\[ = (n \text{Var}(w)) \text{Var}(x). \]

Here we let \( E(w_i) \) be zero. The above equation shows that the variance of the output is the variance of the input, but scaled by \( n \text{Var}(w) \). So if we want to keep the variance of the input and output unchanged, it must have \( n \text{Var}(W) = 1 \), which results in,

\[ \text{Var}(w) = \frac{1}{n}. \]

Hence the weights should be initialized from randomly drawn from a distribution with zero mean and standard variation \( \frac{1}{\sqrt{n}} \).

**ReLU activation: MSR initialization**

When the activation function is \( \text{ReLU}: f(x) = \max(0, x) \), the mean of the input \( E(x) \) is no longer zero. In this case, we have
\[ \text{Var}(y) = \text{Var}\left( \sum_{i}^{n} w_i x_i \right) \]

\[ = \sum_{i}^{n} \text{Var}(w_i x_i) \]

\[ = \sum_{i}^{n} [E(w_i)]^2 \text{Var}(x_i) + E[(x_i)]^2 \text{Var}(w_i) + \text{Var}(x_i) \text{Var}(w_i) \]

\[ = \sum_{i}^{n} \text{Var}(w_i) \{ E[(x_i)]^2 + \text{Var}(x_i) \} \]

\[ = \sum_{i}^{n} \text{Var}(w_i) E(x_i^2) = n \text{Var}(w) E(x^2) . \]

where

\[ E[x^2] = \int_{-\infty}^{+\infty} \max(0, y)^2 p(y) dy \]

\[ = \int_{0}^{+\infty} y^2 p(y) dy \]

\[ = \frac{1}{2} \int_{-\infty}^{+\infty} y^2 p(y) dy \]

\[ = \frac{1}{2} E[(y - E[y])^2] = \frac{1}{2} \text{Var}[y] \]

(see Variance caculation ReLU function) By replacing \( E[x^2] \) by \( \frac{1}{2} \text{Var}[y] \), we have

\[ \text{Var}(y) = n \text{Var}(w) \cdot \frac{1}{2} \text{Var}[y] \]

\[ \Rightarrow \text{Var}(w) = \frac{2}{n} . \]

This is the so called MSR initialization: when we use ReLU as the activation function, the weights should be randomly sampled from distribution with zero mean and \( \frac{2}{n} \) variance.

**Normalization**

Another way to accelerate the training and simplify the optimization process is to use normalization. Normalization usually rescales the data or the weights to make the scale is
just right for the training process. In this section, we will discuss several normalization methods for accelerating and simplifying the training of deep neural networks.

Normalizing the inputs

A common practice to normalize the input data is to compute \( x_i \leftarrow \frac{x_i - \mu}{\sigma} \), where \( \mu \) and \( \sigma \) are the sample mean and standard deviation, respectively. Intuitively, subtracting the mean reduces the shift of the data, and dividing by the standard deviation removes the scale of the data. In statistics, this procedure is called standardizing, and the standardized data is called the z-score.

Another trick is to decorrelate the inputs: if the inputs are uncorrelated, then the weights are independent with each other, which simplifies the problem. One possible way to decorrelated the inputs is to whitening the data with PCA. But sometimes PCA can be harmful for your problem, please see the discussion for further understanding.

Batch (Re)normalization: normalizing the input for each layer

Internal Covariate Shift

Training deep neural networks is difficult due to the changing of the distribution of each layer’s inputs after updating the parameters of the network. It slows down the training by requiring relatively small learning rate and careful weight initialization, especially for networks with saturating nonlinearities. This phenomenon is referred as internal covariate shift.

Batch Normalization

The key is to reduce the internal covariate shift. Motivated by standardizing the input data, we raise such a question: can we normalize the input of each layer? This is exactly what batch normalization (BN) does: it normalize the input of each layer for each training mini-batch. It has two benefits:

1. BN allows higher learning rates and cares less about the initialization.
2. It also acts as a regularizer.

For a layer with \( d \)-dimensional input \( \mathbf{x} = (x^{(1)}, \ldots, x^{(d)}) \), we will normalize each dimension by

\[
\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{Var[x^{(k)}]}},
\]
where $E[x^{(k)}]$ and $\sqrt{Var[x^{(k)}]}$ are the sample mean and standard deviation in a mini-batch. To ensure that the normalized activation keeps the original representation power, BN introduces a pair of parameters,

$$y^{(k)} = \gamma^{(k)} \hat{x}^{(k)} + \beta^{(k)}.$$ We can see that when $\gamma^{(k)} = \sqrt{Var[x^{(k)}]}$ and $\beta^{(k)} = E[x^{(k)}]$ the original activations can be recovered.

It is easy to compute the gradients by chain rule. You may verify your derivation from the BN paper.

Testing/Inference

During training, we compute the mean and variance for each mini-batch. But for testing/inference, we want the output to depend only on the input, deterministically. Recall that, once the training has finished, we can compute the mean and the unbiased variance by using the population. Hence during testing, we use the population mean and unbiased variance estimation instead of the mini-batch mean and variance.

Batch Renormalization

It seems BN has solved everything. E.g., it allows large learning rates, and it is not sensitive to initialization. But what if the batch size is quite small? For small minibatches, the estimates of mean and variance are less accurate. The error accumulates as the depth increases. Moreover, the global estimates of mean and variance are also inaccurate, which may affect the inference/testing accuracy on a large margin!

Let $\mu_B, \sigma_B$ be the minibatch statistics, and $\mu, \sigma$ be their moving averages, then the results of these two normalizations are related by an affine transform,

$$\frac{x_i - \mu}{\sigma} = \frac{x_i - \mu_B + \mu_B - \mu}{\sigma_B} \cdot \frac{\sigma_B}{\sigma} = \frac{x_i - \mu_B}{\sigma_B} \cdot \frac{\sigma_B}{\sigma} + \frac{\mu_B - \mu}{\sigma} = \frac{x_i - \mu_B}{\sigma_B} \cdot r + d,$$

where $r = \frac{\sigma_B}{\sigma}$ and $d = \frac{\mu_B - \mu}{\sigma})$. Batch normalization simply set $r = 1, d = 0$. We refer this augmented batch normalization as Batch Renormalization: the fixed (for the given minibatch) $r$ and $d$ correct for the fact that the minibatch statistics differ from the population ones.

In optimization, we treat $r, d$ as constant to compute the gradients using backpropagation. For more details, please refer to the original paper.

Limitation: for a fixed length feed-forward neural network, BN simply stores the statistics for each layer separately. However, for recurrent neural networks with varied length of sequence as input, applying BN requires different statistics for different time-steps. This limits the
application of BN in recurrent neural networks directly. In the following section, we will see how weight normalization and layer normalization solve this problem.

Normalizing the weights

Instead of normalizing the input of each layer, weight normalization normalizes the weight of each layer by decouple the weight direction and the weight magnitude,

\[ w = g \frac{v}{\|v\|}. \]

Here \( g = \|w\| \) is a scalar, \( v \) is a vector with the same dimensionality of \( w \), and \( \|v\| \) is the Euclidean norm of \( v \). This reparameterization decouples the weight magnitude \( g \) from its direction \( w \).

Normalizing in recurrent neural networks

Layer normalization proposes to reduce the covariate shift problem by fixing the mean and the variance of the summed inputs within each layer.

\[ \mu = \frac{1}{H} \sum_{i=1}^{H} y_i, \quad \sigma^2 = \frac{1}{H} \sum_{i=1}^{H} (y_i - \mu)^2, \]

where \( H \) is the number of activations (neurons), \( y_i = w_i^T x \). In this way, all the hidden neurons in a layer share the same statistics $\mu$, $\sigma^2$, but different training cases have different normalization terms. While for batch normalization, the statistics depend on mini-batches. Hence layer normalization does not impose any constraint on the size of a mini-batch and it can be used in the pure online regime with batch size 1.

Outro

Initialization and normalization are essential for training neural networks. Very deep neural networks, e.g., GoogLeNet and ResNet, are built upon stacks of these methods. Although they seem tricky, they are developed by intuitive motivations and proper assumptions.

As discussed in layer normalization, all the normalization methods can be summarized in the following transformation

\[ y_i = f\left( \frac{g_i}{\sigma_i}(w^T x - \mu_i) + b_i \right), \]

In batch normalization, $\mu_i, \sigma_i$ are computed based on minibatches. While in layer normalization, the statistics are computed by the activations themselves. In weight
normalization, $\mu_i = 0$ and $\sigma_i = ||w||_2$. With careful observations and assumptions, you may develop better initialization/normalization methods.

2 Comments  
deep learning  
🔒  
Login  

Recommend  
Tweet  
Share  
Sort by Best  

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS  

Name

Jae Duk Seo • 2 years ago
amazing post!

rep |  • Reply • Share  

Hongyang Li • 3 years ago
Nice post! So the standard flow is: conv -> norm -> relu -> pooling.
We add the normalization to each conv/fc layer? How about the normalization in the center of a layer?