

DeepGrid

Organic Deep Learning.

Latest Article:

Backpropagation In Convolutional Neural Networks

5 September 2016

[Home](#)

[About](#)

[Archive](#)

[GitHub](#)

[Twitter @jefkine](#)

© 2016. All rights reserved.

Initialization Of Feedforward Networks

Jefkine, 27 July 2016

Mathematics Behind Neural Network Weights Initialization - Part One: In this first of a three part series of posts, we will attempt to go through the weight initialization algorithms as developed by various researchers taking into account influences derived from the evolution of neural network architecture and the activation function in particular.

Introduction

Weight initialization has widely been recognized as one of the most effective approaches in improving training speed of neural networks. Yam and Chow [1] worked on an algorithm for determining the optimal initial weights of feedforward neural networks based on the Cauchy's inequality and a linear algebraic method.

The proposed method aimed at ensuring that the outputs of neurons existed in the **active region** (i.e where the derivative of the activation function has a large value)

while also increasing the rate of convergence. From the outputs of the last hidden layer and the given output patterns, the optimal values of the last layer of weights are evaluated by a least-squares method.

With the optimal initial weights determined, the initial error becomes substantially smaller and the number of iterations required to achieve the error criterion is significantly reduced.

Yam and Chow had previously worked on other methods with this method largely seen as an improvement on earlier proposed methods [2] and [3]. In [1] the rationale behind their proposed approach was to reduce the initial network error while preventing the network from getting stuck with the initial weights.

Weight Initialization Method

Consider a multilayer neural network with L fully interconnected layers. Layer l consists of $n_l + 1$ neurons ($l = 1, 2, \dots, L - 1$) in which the last neuron is a bias node with a constant output of 1.

Notation

1. l is the l^{th} layer where $l = 1$ is the first layer and $l = L$ is the last layer.
2. $o_{i,j}^l$ is the output vector at layer l

$$o_{i,j}^l = \sum_{i=1}^{n_l+1} w_{i \rightarrow j}^l a_{p,i}^l$$

3. $w_{i \rightarrow j}^l$ is the weight vector connecting neuron i of layer l with neuron j of layer $l + 1$.
4. a^l is the activated output vector for a hidden layer l .

$$a_{p,j}^l = f(o_{p,j}^{l-1})$$

5. $f(x)$ is the activation function. A sigmoid function with the range of between 0 and 1 is used as the activation function.

$$f(x) = \frac{1}{1 + e^{(-x)}}$$

6. X^1 where $l = 1$ is the matrix of all given inputs with dimensions P rows and $n_l + 1$ columns. All the last entries of the matrix X^1 are a constant 1.
7. W^l is the weight matrix between layers l and $l + 1$.
8. A^L is the matrix representing all entries of the last output layer L given by

$$a_{p,j}^L = f(o_{p,j}^{L-1})$$

9. P is the matrix representing all the training patterns for network training
10. T is the matrix of all the targets with dimensions P rows and n_L columns.

The output of all hidden layers and the output layer are obtained by propagating the training patterns through the network.

Learning will then be achieved by adjusting the weights such that A^L is as close as possible or equals to T .

In the classical back-propagation algorithm, the weights are changed according to the gradient descent direction of an error surface E (taken on the output units over all the P patterns) using the following formula:

$$E_p = \frac{1}{2} \sum_{p=1}^P \left(\sum_j^{n_L} (t_{p,j} - a_{p,j})^2 \right)$$

Hence, the error gradient of the weight matrix $w_{i \rightarrow j}$ is:

$$\begin{aligned} \frac{\partial E_p}{\partial w_{i \rightarrow j}^l} &= \frac{\partial E_p}{\partial o_{i,j}^l} \frac{\partial o_{i,j}^l}{\partial w_{i \rightarrow j}^l} \\ &= \delta_{p,j}^l \frac{\partial}{\partial w_{i \rightarrow j}^l} w_{i \rightarrow j}^l a_{p,i}^l \\ &= \delta_{p,j}^l a_{p,i}^l \end{aligned}$$

If the standard sigmoid function with a range between 0 and 1 is used, the rule of changing the weights can be shown to be:

$$\Delta w_{i,j}^l = - \frac{\eta}{P} \sum_{p=1}^P \delta_{p,j}^l a_{p,i}^l$$

where η is the learning rate (or rate of gradient descent)

The error gradient of the input vector at a layer l is defined as

$$\delta_{p,j}^l = \frac{\partial E_p}{\partial o_{p,j}^l}$$

The error gradient of the input vector at the last layer $l = L - 1$ is

$$\begin{aligned}
\delta_{p,j}^{L-1} &= \frac{\partial E_p}{\partial o_{p,j}^{L-1}} \\
&= \frac{\partial}{\partial o_{p,j}^{L-1}} \frac{1}{2} (t_{p,j} - a_{p,j}^L)^2 \\
&= \left(\frac{\partial}{\partial a_{p,j}^L} \frac{1}{2} (t_{p,j} - a_{p,j}^L)^2 \right) \frac{\partial a_{p,j}^L}{\partial o_{p,j}^{L-1}} \\
&= (t_{p,j} - a_{p,j}^L) \frac{\partial f(o_{p,j}^{L-1})}{\partial o_{p,j}^{L-1}} \\
&= (t_{p,j} - a_{p,j}^L) f'(o_{p,j}^{L-1})
\end{aligned} \tag{1}$$

Derivative of a sigmoid $f'(o^{L-1}) = f(o^{L-1})(1 - f(o^{L-1}))$. This result can be substituted in equation (1) as follows:

$$\begin{aligned}
\delta_{p,j}^{L-1} &= (t_{p,j} - a_{p,j}^L) f(o_{p,j}^{L-1}) (1 - f(o_{p,j}^{L-1})) \\
&= (t_{p,j} - a_{p,j}^L) a_{p,j}^L (1 - a_{p,j}^L)
\end{aligned} \tag{2}$$

For the other layers i.e $l = 1, 2, \dots, L - 2$, the scenario is such that multiple weighted outputs from previous the layer $l - 1$ lead to a unit in the current layer l yet again multiple outputs.

The weight $w_{j \rightarrow k}^{l+1}$ connects neurons j and k , the sum of the weighed inputs from neuron j is denoted by $k \in \text{outs}(j)$ where k iterates over all neurons connected to j

$$\begin{aligned}
\delta_{p,j}^l &= \frac{\partial E_p}{\partial o_{p,j}^l} \\
&= \frac{\partial E_p}{\partial o_{j,k}^{l+1}} \frac{\partial o_{j,k}^{l+1}}{\partial o_{p,j}^l} \\
&= \delta_{p,k}^{l+1} \frac{\partial o_{j,k}^{l+1}}{\partial o_{p,j}^l} \\
&= \delta_{p,k}^{l+1} \frac{\partial w_{j \rightarrow k}^{l+1} a_{p,j}^{l+1}}{\partial o_{p,j}^l} \\
&= \delta_{p,k}^{l+1} \frac{\partial w_{j \rightarrow k}^{l+1} a_{p,j}^{l+1}}{\partial a_{p,j}^{l+1}} \frac{\partial a_{p,j}^{l+1}}{\partial o_{p,j}^l} \\
&= \delta_{p,k}^{l+1} \frac{\partial w_{j \rightarrow k}^{l+1} a_{p,j}^{l+1}}{\partial a_{p,j}^{l+1}} \frac{\partial f(o_{p,j}^l)}{\partial o_{p,j}^l} \\
&= \frac{\partial f(o_{p,j}^l)}{\partial o_{p,j}^l} \sum_{k \in \text{outs}(j)} \delta_{p,k}^{l+1} \frac{\partial}{\partial a_{p,j}^{l+1}} w_{j \rightarrow k}^{l+1} a_{p,j}^{l+1} \\
&= f'(o_{p,j}^l) \sum_{k \in \text{outs}(j)} \delta_{p,k}^{l+1} w_{j \rightarrow k}^{l+1}
\end{aligned} \tag{3}$$

Derivative of a sigmoid $f'(o^l) = f(o^l)(1 - f(o^l))$. This result can be substituted in equation (3) as follows:

$$\begin{aligned}
\delta_{p,j}^l &= f(o_{p,j}^l)(1 - f(o_{p,j}^l)) \sum_{k \in \text{outs}(j)} \delta_{p,k}^{l+1} w_{j \rightarrow k}^{l+1} \\
&= a_{p,j}^{l+1}(1 - a_{p,j}^{l+1}) \sum_{k \in \text{outs}(j)} \delta_{p,k}^{l+1} w_{j \rightarrow k}^{l+1}
\end{aligned} \tag{4}$$

From Eqns. (2) and (4), we can observe that the change in weight depends on outputs of neurons connected to it. When outputs of neurons are 0 or 1, the derivative of the activation function evaluated at this value is zero. This means there will be no weight change at all even if there is a difference between the value of the target and the actual output.

The magnitudes required to ensure that the outputs of the hidden units are in the active region and are derived by solving the following problem:

$$1 - \bar{t} \leq a_{p,j}^{l+1} \leq \bar{t} \tag{6}$$

A sigmoid function also has the **property** $f(1 - x) = f(-x)$ which means Eqn (6) can also be written as:

$$-\bar{t} \leq a_{p,j}^{l+1} \leq \bar{t} \quad (7)$$

Taking the inverse of Eqn(7) such that $f^{-1}(\bar{t}) = s$ and $f^{-1}(a_{p,j}^{l+1}) = o_{p,j}^l$ results in

$$-\bar{s} \leq o_{p,j}^l \leq \bar{s} \quad (8)$$

The active region is then assumed to be the region in which the derivative of the activation function is greater than 4 of the maximum derivative, i.e.

$$\bar{s} \approx 4.59 \quad \text{for the standard sigmoid function} \quad (9a)$$

and

$$\bar{s} \approx 2.29 \quad \text{for the hyperbolic tangent function} \quad (9b)$$

Eqn(8) can then be simplified to

$$(o_{p,j}^l)^2 \leq \bar{s}^2 \quad \text{or} \quad \left(\sum_{i=1}^{n_l+1} a_{p,i}^l w_{i,j}^l \right)^2 \leq \bar{s}^2 \quad (10)$$

By Cauchy's inequality,

$$\left(\sum_{i=1}^{n_l+1} a_{p,i}^l w_{i,j}^l \right)^2 \leq \sum_{i=1}^{n_l+1} (a_{p,i}^l)^2 \sum_{i=1}^{n_l+1} (w_{i,j}^l)^2 \quad (11)$$

Consequently, Eqn(10) is replaced by

$$\sum_{i=1}^{n_l+1} (a_{p,i}^l)^2 \sum_{i=1}^{n_l+1} (w_{i,j}^l)^2 \leq \bar{s}^2 \quad (12)$$

If n_l is a large number and if the weights are values between $-\theta_p^l$ to θ_p^l with zero mean independent identical distributions, the general weight formula becomes:

$$\sum_{i=1}^{n_l+1} (w_{i,j}^l)^2 \approx \left[\begin{array}{c} \text{number} \\ \text{of} \\ \text{weights} \end{array} \right] * \left[\begin{array}{c} \text{variance} \\ \text{of} \\ \text{weights} \end{array} \right] \quad (13)$$

For uniformly distributed sets of weights $w^l \stackrel{iid}{\sim} U[-\theta_p^l, \theta_p^l]$ we apply the formula $\left\{ x \sim U[a, b] \implies Var[x] = \frac{(b-a)^2}{12} \right\}$ for [variance of a uniform distribution](#), and subsequently show that.

$$\begin{aligned} Var[w^l] &= \frac{(2\theta_p^l)^2}{12} \\ Var[w^l] &= \frac{(\theta_p^l)^2}{3} \end{aligned}$$

This can then be used in the next Eqn as:

$$\sum_{i=1}^{n_l+1} (w_{i,j}^l)^2 \approx (n_l + 1) * \frac{(\theta_p^l)^2}{3} \approx \frac{(n_l + 1)}{3} (\theta_p^l)^2 \quad (14)$$

By substituting Eqn(14) into (12)

$$\begin{aligned} \sum_{i=1}^{n_l+1} (a_{p,i}^l)^2 \frac{(n_l + 1)}{3} (\theta_p^l)^2 &\leq \bar{s}^2 \\ (\theta_p^l)^2 &\leq \bar{s}^2 \left[\frac{3}{(n_l + 1) \left(\sum_{i=1}^{n_l+1} (a_{p,i}^l)^2 \right)} \right] \\ \theta_p^l &\leq \bar{s} \sqrt{\frac{3}{(n_l + 1) \left(\sum_{i=1}^{n_l+1} (a_{p,i}^l)^2 \right)}} \quad (15) \end{aligned}$$

For sets of weights with normal distribution $w^l \stackrel{iid}{\sim} \mathcal{N}(0, (\theta_p^l)^2)$ and n_l is a large number, $\{\sigma^2 = (\theta_p^l)^2 \implies Var[w^l] = (\theta_p^l)^2\}$

$$\sum_{i=1}^{n_l+1} (w_{i,j}^l)^2 \approx (n_l + 1) * (\theta_p^l)^2 \approx \frac{(n_l + 1)}{1} (\theta_p^l)^2 \quad (16)$$

By substituting Eqn(16) into (12)

$$\begin{aligned} \sum_{i=1}^{n_l+1} (a_{p,i}^l)^2 \frac{(n_l + 1)}{1} (\theta_p^l)^2 &\leq \bar{s}^2 \\ (\theta_p^l)^2 &\leq \bar{s}^2 \left[\frac{1}{(n_l + 1) \left(\sum_{i=1}^{n_l+1} (a_{p,i}^l)^2 \right)} \right] \\ \theta_p^l &\leq \bar{s} \sqrt{\frac{1}{(n_l + 1) \left(\sum_{i=1}^{n_l+1} (a_{p,i}^l)^2 \right)}} \quad (17) \end{aligned}$$

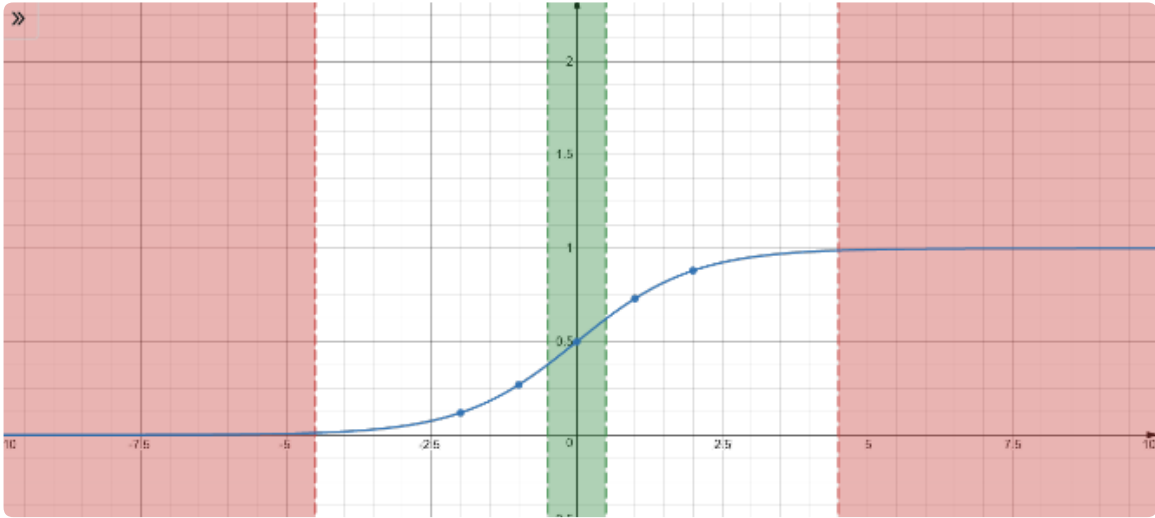
In the proposed algorithm therefore, the magnitude of weights θ_p^l for pattern p is chosen to be:

$$\theta_p^l \leq \begin{cases} \frac{1}{s} \sqrt{\frac{3}{(n_l+1) \left(\sum_{i=1}^{n_l+1} (a_{p,i}^l)^2 \right)}}, & \text{for weights with uniform distribution} \\ \frac{1}{s} \sqrt{\frac{1}{(n_l+1) \left(\sum_{i=1}^{n_l+1} (a_{p,i}^l)^2 \right)}}, & \text{for weights with normal distribution} \end{cases} \quad (18)$$

For different input patterns, the values of θ_p^l are different, To make sure the outputs of hidden neurons are in the active region for all patterns, the following value is selected

$$\theta^l = \min_{p=1, \dots, P} (\theta_p^l) \quad (19)$$

Deep networks using of sigmoid activation functions exhibit the challenge of exploding or vanishing activations and gradients. The diagram below gives us a perspective of how this occurs.



- When the parameters are too large - the activations become larger and larger (i.e tend to exist in the red region above), then your activations will saturate and become meaningless, with gradients approaching 0.
- When the parameters are too small - the activations keep dropping layer after layer (i.e tend to exist in the green region above). At levels closer to zero the sigmoid activation function become more linear. Gradually the non-linearity is lost with derivatives of constants being 0 and hence no benefit of multiple layers.

Summary Procedure of The Weight Initialization Algorithm

1. Evaluate θ^1 using the input training patterns by applying Eqns (18) and (19) with $l = 1$
2. The weights $w_{i \rightarrow j}^1$ are initialized by a random number generator with uniform distribution between $-\theta^1$ to θ^1 or normal distribution $\mathcal{N}(0, (\theta_p^1)^2)$
3. Evaluate $a_{p,i}^2$ by feedforwarding the input patterns through the network using $w_{i \rightarrow j}^1$
4. For $l = 1, 2, \dots, L - 2$
 - Evaluate θ^l using the outputs of layer l , i.e $a_{p,i}^l$ and applying Eqns (18) and (19)
 - The weights $w_{i \rightarrow j}^l$ are initialized by a random number generator with uniform distribution between $-\theta^l$ to θ^l or normal distribution $\mathcal{N}(0, (\theta_p^l)^2)$
 - Evaluate $a_{p,i}^{l+1}$ by feedforwarding the outputs of $a_{p,i}^l$ through the network using $w_{i \rightarrow j}^l$
5. After finding $a_{p,i}^{L-1}$ or A^{L-1} , we can find the last layer of weights W^{L-1} by solving the following equation using a least-squares method,

$$\|A^{L-1}W^{L-1} - S\|_2 \quad (20)$$

- S is a matrix, which has entries,

$$s_{i,j} = f^{-1}(t_{i,j}) \quad (21)$$

- where $t_{i,j}$ are the entries of target matrix T .

The weight initialization process is then completed. The linear least-squares problem shown in Eqn(20) can be solved by QR factorization using Householder reflections [4].

In the case of an overdetermined system, QR factorization produces a solution that is the best approximation in a least-squares sense. In the case of an under-determined system, QR factorization computes the minimal-norm solution.

Conclusions

In general the proposed algorithm with uniform distributed weights performs better than the algorithm with normal distributed weights. The proposed algorithm is also applicable to networks with different activation functions.

It is worth noting that the time required for the initialization process is negligible when compared with training process.

References

1. A weight initialization method for improving training speed in feedforward neural network Jim Y.F. Yam, Tommy W.S. Chow (1998) [\[pdf\]](#)
2. Y.F. Yam, T.W.S. Chow, Determining initial weights of feedforward neural networks based on least-squares method, Neural Process. Lett. 2 (1995) 13-17.
3. Y.F. Yam, T.W.S. Chow, A new method in determining the initial weights of feedforward neural networks, Neurocomputing 16 (1997) 23-32.
4. G.H. Golub, C.F. Van Loan, Matrix Computations, The Johns Hopkins University Press, Baltimore, MD, 1989

Related Posts

Initialization Of Deep Feedfoward Networks 01 Aug 2016

Initialization Of Deep Networks Case of Rectifiers 08 Aug 2016

0 Comments

Deep Grid

 Login ▾

♥ Recommend 1

 Share

Sort by Best ▾



Start the discussion...

Be the first to comment.

 Subscribe  Add Disqus to your site Add Disqus Add  Privacy

DeepGrid

Organic Deep Learning.

Latest Article:

Backpropagation In Convolutional Neural Networks

5 September 2016

[Home](#)

[About](#)

[Archive](#)

[GitHub](#)

[Twitter @jefkine](#)

© 2016. All rights reserved.

Initialization Of Deep Feedforward Networks

Jefkine, 1 August 2016

Mathematics Behind Neural Network Weights Initialization - Part Two: In this second of a three part series of posts, we will attempt to go through the weight initialization algorithms as developed by various researchers taking into account influences derived from the evolution of neural network architecture and the activation function in particular.

Introduction

Deep multi-layered neural networks often require experiments that interrogate different initialization routines, activations and variation of gradients across layers during training. These provide valuable insights into what aspects should be improved to aid faster and successful training.

For Xavier and Bengio (2010) [1] the objective was to better understand why standard gradient descent from random initializations was performing poorly in deep neural networks. They carried out analysis driven by investigative experiments that monitored activations (watching for saturations of hidden units) and gradients across layers and across training iterations. They evaluated effects on choices of different activation functions (how it might affect saturation) and initialization procedure (with lessons learned from unsupervised pre-training as a form of initialization that already had drastic impact)

A new initialization scheme that brings substantially faster convergence was proposed. In this article we discuss the algorithm put forward by Xavier and Bengio (2010) [1]

Gradients at Initialization

Earlier on, Bradley (2009) [2] found that in networks with linear activation at each layer, the variance of the back-propagated gradients decreases as we go backwards in the network. Below we will look at theoretical considerations and a derivation of the **normalized initialization**.

Notation

1. f is the activation function. For the dense artificial neural network, a symmetric activation function f with unit derivative at 0 (*i.e.* $f'(0) = 1$) is chosen. Hyperbolic tangent and softsign are both forms of symmetric activation functions.
2. z^i is the activation vector at layer i , $z^i = f(s^{i-1})$
3. s^i is the argument vector of the activation function at layer i ,
$$s^i = z^{i-1}w^i + b^i$$
4. w^i is the weight vector connecting neurons in layer i with neurons in layer $i + 1$.
5. b^i is the bias vector on layer i .
6. x is the network input.

From the notation above, it is easy to derive the following equations of back-propagation

$$\frac{\partial Cost}{\partial s_k^i} = f'(s_k^i) \sum_{l \in \text{outs}(k)} (w_{k \rightarrow l}^{i+1}) \frac{\partial Cost}{\partial s_k^{i+1}} \quad (1a)$$

$$\frac{\partial Cost}{\partial s_k^i} = f'(s_k^i) W_{k,\bullet}^{i+1} \frac{\partial Cost}{\partial s_k^{i+1}} \quad (1b)$$

$$\frac{\partial Cost}{\partial w_{l,k}^i} = z_l^i \frac{\partial Cost}{\partial s_k^i} \quad (2)$$

The variances will be expressed with respect to the input, output and weight initialization randomness. Considerations made include:

- Initialization occurs in a linear regime
- Weights are initialized independently
- Input feature variances are the same ($= \text{Var}[x]$)
- There is no correlation between our input and our weights and both are zero-mean.
- All biases have been initialized to zero.

For the input layer, $X \in \mathbb{R}^{m \times n}$ with n components each from m training samples. Here the neurons are linear with random weights $W^{(in \rightarrow 1)} \in \mathbb{R}^{m \times a}$ outputting $S^{(1)} \in \mathbb{R}^{n \times a}$.

The output $S^{(1)}$ can be shown by the equations below:

$$S^{(1)} = XW^{(in \rightarrow 1)}$$

$$S_{ij}^{(1)} = \sum_{k=1}^m X_{ik} W_{kj}^{(in \rightarrow 1)}$$

Given X and W are independent, we can show that the **variance of their product** can be given by:

$$\text{Var}(W_i X_i) = E[X_i]^2 \text{Var}(W_i) + E[W_i]^2 \text{Var}(X_i) + \text{Var}(W_i) \text{Var}(X_i) \quad (3)$$

Considering our inputs and weights both have mean 0, Eqn. (3) simplifies to

$$\text{Var}(W_i X_i) = \text{Var}(W_i) \text{Var}(X_i)$$

X_i and W_i are all independent and identically distributed, **we can therefore show that:**

$$\text{Var}\left(\sum_{i=1}^n W_i X_i\right) = \text{Var}(W_1 X_1 + W_2 X_2 + \dots + W_n X_n) = n \text{Var}(W_i) \text{Var}(X_i)$$

Further, lets now look at two adjacent layers i and i' . Here, n_i is used to denote the size of layer i . Applying the derivative of the activation function at s_k^i yields a

value of approximately one.

$$f'(s_k^i) \approx 1, \quad (4)$$

Then using our prior knowledge of independent and identically distributed X_i and W_i , we have.

$$Var[z^i] = Var[x] \prod_{i'=0}^{i-1} n_{i'} Var[W^{i'}] \quad (5)$$

$Var[W^{i'}]$ in Eqn. 5, is the shared scalar variance of all weights at layer i' . Taking these observations into consideration, a network with d layers, will have the following Eqns.

$$Var \left[\frac{\partial Cost}{\partial s^i} \right] = Var \left[\frac{\partial Cost}{\partial s^d} \right] \prod_{i'=i}^d n_{i'+1} Var[W^{i'}], \quad (6)$$

$$Var \left[\frac{\partial Cost}{\partial w^i} \right] = \prod_{i'=0}^{i-1} n_{i'} Var[W^{i'}] \prod_{i'=i}^{d-1} n_{i'+1} Var[W^{i'}] \times Var[x] Var \left[\frac{\partial Cost}{\partial s^d} \right]. \quad (7)$$

We would then like to steady the variance such there is equality from layer to layer. From a forward-propagation point of view, to keep information flowing we would like that

$$\forall(i, i'), Var[z^i] = Var[z^{i'}]. \quad (8)$$

From a back-propagation point of view, we would like to have:

$$\forall(i, i'), Var \left[\frac{\partial Cost}{\partial s^i} \right] = Var \left[\frac{\partial Cost}{\partial s^{i'}} \right]. \quad (9)$$

For Eqns. (8) and (9) to hold, the shared scalar variances $n_{i'} Var[W^{i'}]$ in Eqn. (5) should be 1. This is the same as trying to ensure the variances of the input and output are consistent (realize that the technique used here helps avoid reducing or magnifying the signals exponentially hence mitigating the exploding or vanishing gradient problem):

$$\forall(i), \quad n_i Var[W^i] = 1 \quad (10a)$$

$$\forall(i), \quad Var[W^i] = \frac{1}{n_i} \quad (10b)$$

$$\forall(i), \quad n_{i+1} Var[W^i] = 1 \quad (11a)$$

$$\forall(i), \quad Var[W^i] = \frac{1}{n_{i+1}} \quad (11b)$$

As a compromise between the two constraints (representing back-propagation and forward-propagation), we might want to have

$$\forall(i), \quad \frac{2}{n_i + n_{i+1}} \quad (12)$$

In the experimental setting chosen by Xavier and Bengio (2010) [1], the standard initialization weights $W_{i,j}$ at each layer using the commonly used heuristic:

$$W_{i,j} \sim U \left[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}} \right], \quad (13)$$

where $U [-\theta, \theta]$ is the uniform distribution in the interval $(-\theta, \theta)$ and n is the size of the previous layer (the number of columns of W).

For uniformly distributed sets of weights $W \stackrel{iid}{\sim} U [-\theta, \theta]$ with zero mean, we can use the formula $\left\{ x \sim U[a, b] \implies Var[x] = \frac{(b-a)^2}{12} \right\}$ for [variance of a uniform distribution](#) to show that:

$$\begin{aligned} Var[W] &= \frac{(2\theta)^2}{12} \\ Var[W] &= \frac{\theta^2}{3} \end{aligned} \quad (14)$$

Substituting Eqn. (14) into an Eqn. of the form $nVar[W] = 1$ yields:

$$\begin{aligned} n \frac{\theta^2}{3} &= 1 \\ \theta^2 &= \frac{3}{n} \\ \theta &= \frac{\sqrt{3}}{\sqrt{n}} \end{aligned}$$

The weights have thus been initialized from the uniform distribution over the interval

$$W \sim U \left[-\frac{\sqrt{3}}{\sqrt{n}}, \frac{\sqrt{3}}{\sqrt{n}} \right] \quad (15)$$

The normalization factor may therefore be important when initializing deep networks because of the multiplicative effect through layers. The suggestion then is of an initialization procedure that maintains stable variances of activation and back-propagated gradients as one moves up or down the network. This is known as the **normalized initialization**

$$W \sim U \left[-\frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}}, \frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}} \right] \quad (16)$$

The normalized initialization is a clear compromise between the two constraints involving n_i and n_{i+1} (representing back-propagation and forward-propagation), If you used the input $X \in \mathbb{R}^{m \times n}$, the normalized initialization would look as follows:

$$W \sim U \left[-\frac{\sqrt{6}}{\sqrt{n + m}}, \frac{\sqrt{6}}{\sqrt{n + m}} \right] \quad (17)$$

Conclusions

In general, from Xavier and Bengio (2010) [1] experiments we can see that the variance of the gradients of the weights is the same for all the layers, but the variance of the back-propagated gradient might still vanish or explode as we consider deeper networks.

Applications

The initialization routines derived here, more famously known as “**Xavier Initialization**” have been successfully applied in various deep learning libraries. Below we shall look at [Keras](#) a minimalist, highly modular neural networks library, written in Python and capable of running on top of either TensorFlow or Theano.

The initialization routine here is named “glorot_” following the name of one of the authors Xavier Glorot [1]. In the code snippet below, **glorot_normal** is the implementation of Eqn. (12) while **glorot_uniform** is the equivalent implementation of Eqn. (15)

```
def get_fans(shape):
    fan_in = shape[0] if len(shape) == 2 else np.prod(shape[1:])
    fan_out = shape[1] if len(shape) == 2 else shape[0]
    return fan_in, fan_out

def glorot_normal(shape, name=None):
    ''' Reference: Glorot & Bengio, AISTATS 2010
    '''
    fan_in, fan_out = get_fans(shape)
    s = np.sqrt(2. / (fan_in + fan_out))
    return normal(shape, s, name=name)

def glorot_uniform(shape, name=None):
    fan_in, fan_out = get_fans(shape)
    s = np.sqrt(6. / (fan_in + fan_out))
    return uniform(shape, s, name=name)
```


References

1. Glorot Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." Aistats. Vol. 9. 2010. [\[pdf\]](#)
2. Bradley, D. (2009). Learning in modular systems. Doctoral dissertation, The Robotics Institute, Carnegie Mellon University.
3. Wikipedia - "Product of independent variables". [Variance](#).

Related Posts

Initialization Of Feedfoward Networks 27 Jul 2016

Initialization Of Deep Networks Case of Rectifiers 08 Aug 2016

0 Comments

Deep Grid

 Login ▾

 Recommend



 Share

Sort by Best ▾



Start the discussion...

Be the first to comment.

 Subscribe  Add Disqus to your site [Add Disqus Add](#)  Privacy

DeepGrid

Organic Deep Learning.

Latest Article:

Backpropagation In Convolutional Neural Networks

5 September 2016

[Home](#)

[About](#)

[Archive](#)

[GitHub](#)

[Twitter @jefkine](#)

© 2016. All rights reserved.

Initialization Of Deep Networks Case of Rectifiers

Jefkine, 8 August 2016

Mathematics Behind Neural Network Weights Initialization - Part Three: In this third of a three part series of posts, we will attempt to go through the weight initialization algorithms as developed by various researchers taking into account influences derived from the evolution of neural network architecture and the activation function in particular.

Introduction

Recent success in deep networks can be credited to the use of non-saturated activation function Rectified Linear Unit (ReLU) which has replaced its saturated counterpart (e.g. sigmoid, tanh). Benefits associated with the ReLU activation function include:

- Ability to mitigate the exploding or vanishing gradient problem; this largely due to the fact that for all inputs into the activation function of values greater than 0, the gradient is always 1 (constant gradient)
- The constant gradient of ReLUs results in faster learning. This helps expedite convergence of the training procedure yielding better solutions than sigmoidlike units
- Unlike the sigmoidlike units (such as sigmoid or tanh activations) ReLU activations does not involve computing of an exponent which is a factor that results to a faster training and evaluation times.
- Producing sparse representations with true zeros. All inputs into the activation function of values less than or equal to 0, results in an output value of 0. Sparse representations are considered more valuable

Xavier and Bengio (2010) [2] had earlier on proposed the “Xavier” initialization, a method whose derivation was based on the assumption that the activations are linear. This assumption however is invalid for ReLU and PReLU. He, Kaiming, et al. 2015 [1] later on derived a robust initialization method that particularly considers the rectifier nonlinearities.

In this article we discuss the algorithm put forward by He, Kaiming, et al. 2015 [1].

Notation

1. k is the side length of a convolutional kernel. (also the spatial filter size of the layer)
2. c is the channel number. (also input channel)
3. n is the number of connections of a response ($n = k \times k \times c \implies k^2 c$)
4. \mathbf{x} is the co-located $k \times k$ pixels in c inputs channels. ($\mathbf{x} = (k^2 c) \times 1$)
5. W is the matrix where d is the total number number of filters. Every row of W i.e (d_1, d_2, \dots, d_d) represents the weights of a filter. ($W = d \times n$)
6. \mathbf{b} is the vector of biases.
7. \mathbf{y} is the response pixel of the output map
8. l is used to index the layers
9. $f(\cdot)$ is the activation function
10. $\mathbf{x} = f(\mathbf{y}_{l-1})$
11. $c = d_{l-1}$
12. $\mathbf{y}_l = W_l \mathbf{x}_l + \mathbf{b}_l$

Forward Propagation Case

Considerations made here include:

- The initialized elements in W_l be mutually independent and share the same distribution.
- The elements in \mathbf{x}_l are mutually independent and share the same distribution.
- \mathbf{x}_l and W_l are independent of each other.

The variance of y_l can be given by:

$$Var[y_l] = n_l Var[w_l x_l], \quad (1)$$

where y_l, w_l, x_l represent random variables of each element in $\mathbf{y}_l, W_l, \mathbf{x}_l$. Let w_l have a zero mean, then the variance of the product of independent variables gives us:

$$Var[y_l] = n_l Var[w_l] E[x_l^2]. \quad (2)$$

Lets look at how the Eqn. (2) above is arrived at:-

For random variables \mathbf{x}_l and W_l , independent of each other, we can use basic properties of expectation to show that:

$$Var[w_l x_l] = E[w_l^2] E[x_l^2] - \overbrace{[E[x_l]]^2 [E[w_l]]^2}^{\star}, \quad (A)$$

From Eqn. (2) above, we let w_l have a zero mean $\implies E[w_l] = [E[w_l]]^2 = 0$.

This means that in Eqn. (A), \star evaluates to zero. We are then left with:

$$Var[w_l x_l] = E[w_l^2] E[x_l^2], \quad (B)$$

Using the formula for **variance** $Var[w_l] = E[w_l^2] - [E[w_l]]^2$ and the fact that $E[w_l] = 0$ we come to the conclusion that $Var[w_l] = E[w_l^2]$.

With this conclusion we can replace $E[w_l^2]$ in Eqn. (B) with $Var[w_l]$ to obtain the following Eqn.:

$$Var[w_l x_l] = Var[w_l] E[x_l^2], \quad (C)$$

By substituting Eqn. (C) into Eqn. (1) we obtain:

$$Var[y_l] = n_l Var[w_l] E[x_l^2]. \quad (2)$$

In Eqn. (2) it is well worth noting that $E[x_l^2]$ is the expectation of the square of x_l and cannot resolve to $Var[x_l]$ i.e $E[x_l^2] \neq Var[x_l]$ as we did above for w_l unless x_l has zero mean.

The effect of ReLU activation is such that $x_l = \max(0, y_{l-1})$ and thus it does not have zero mean. For this reason the conclusion here is different compared to the initialization style in [2].

We can also observe here that despite the mean of x_l i.e $E[x_l]$ being non zero, the product of the two means $E[x_l]$ and $E[w_l]$ will lead to a zero mean since $E[w_l] = 0$ as shown in the Eqn. below:

$$E[y_l] = E[w_l x_l] = E[x_l] E[w_l] = 0.$$

If we let w_{l-1} have a symmetric distribution around zero and $b_{l-1} = 0$, then from our observation above y_{l-1} has zero mean and a symmetric distribution around zero. This leads to $E[x_l^2] = \frac{1}{2} \text{Var}[y_{l-1}]$ when $f(\cdot)$ is ReLU. Putting this in Eqn. (2), we obtain:

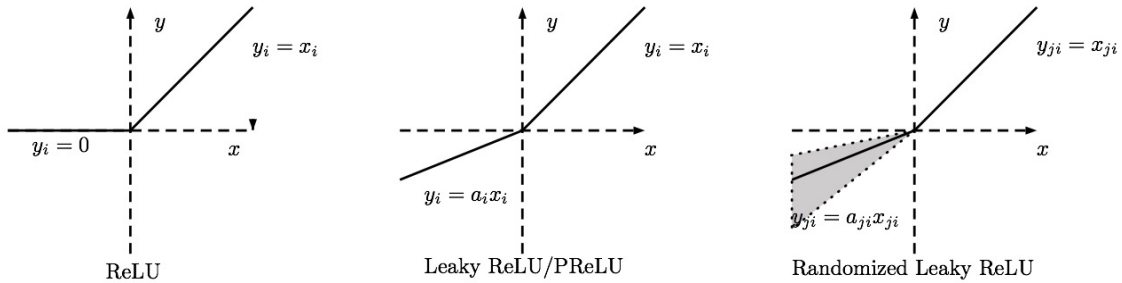
$$\text{Var}[y_l] = n_l \text{Var}[w_l] \frac{1}{2} \text{Var}[y_{l-1}]. \quad (3)$$

With L layers put together, we have:

$$\text{Var}[y_L] = \text{Var}[y_1] \left(\prod_{l=1}^L \frac{1}{2} n_l \text{Var}[w_l] \right). \quad (4)$$

The product in Eqn. (4) is key to the initialization design.

Lets take some time to explain the effect of ReLU activation as seen in Eqn. (3).



For the family of rectified linear (ReL) shown illustrated in the diagram above, we have a generic activation function defined as follows:

$$f(y_i) = \begin{cases} y_i, & \text{if } y_i > 0 \\ a_i y_i, & \text{if } y_i \leq 0 \end{cases} \quad (5)$$

In the activation function (5) above, y_i is the input of the nonlinear activation f on the i th channel, and a_i is the coefficient controlling the slope of the negative part. i in a_i indicates that we allow the nonlinear activation to vary on different channels.

The variations of rectified linear (ReL) take the following forms:

1. **ReLU**: obtained when $a_i = 0$. The resultant activation function is of the form

$$f(y_i) = \max(0, y_i)$$
2. **PReLU**: Parametric ReLU - obtained when a_i is a learnable parameter. The resultant activation function is of the form

$$f(y_i) = \max(0, y_i) + a_i \min(0, y_i)$$
3. **LReLU**: Leaky ReLU - obtained when $a_i = 0.01$ i.e when a_i is a small and fixed value [3]. The resultant activation function is of the form

$$f(y_i) = \max(0, y_i) + 0.01 \min(0, y_i)$$
4. **RReLU**: Randomized Leaky ReLU - the randomized version of leaky ReLU, obtained when a_{ji} is a random number sampled from a uniform distribution $U(l, u)$ i.e $a_{ji} \sim U(l, u)$; $l < u$ and $l, u \in [0; 1)$. See [4].

Rectifier activation function is simply a threshold at zero hence allowing the network to easily obtain sparse representations. For example, after uniform initialization of the weights, around 50% of hidden units continuous output values are real zeros, and this fraction can easily increase with sparsity-inducing regularization [5].

Take signal $y_i = W_i x + b$, (visualize the signal represented on a bi-dimensional space $y_i \in \mathbb{R}^2$). Applying the rectifier activation function to this signal i.e $f(y_i)$ where f is ReLU results in a scenario where signals existing in regions where $y_i < 0$ are squashed to 0, while those existing in regions where $y_i > 0$ remain unchanged.

The ReLU effect results in “*aggressive data compression*” where information is lost (replaced by real zeros values). A remedy for this would be the PReLU and LReLU implementations which provides an axle shift that adds a slope a_i to the negative section ensuring from the data some information is retained rather than reduced to zero. Both PReLU and LReLU represented by variations of $f(y_i) = \max(0, y_i) + a_i \min(0, y_i)$, make use of the factor a_i which serves as the component used to retain some information.

Using ReLU activation function therefore, only the positive half axis values are obtained hence:

$$E[x_l^2] = \frac{1}{2} \text{Var}[y_{l-1}] \quad (6)$$

Putting this in Eqn. (2), we obtain our Eqn. (3) as above:

$$Var[y_l] = n_l Var[w_l] \frac{1}{2} Var[y_{l-1}] \quad (3a)$$

$$= \frac{1}{2} n_l Var[w_l] Var[y_{l-1}] \quad (3b)$$

Note that a proper initialization method should avoid reducing and magnifying the magnitudes of input signals exponentially. For this reason we expect the product in Eqn. (4) to take a proper scalar (e.g., 1). This leads to:

$$\frac{1}{2} n_l Var[w_l] = 1, \quad \forall l \quad (7)$$

From Eqn. (7) above, we can conclude that:

$$Var[w_l] = \frac{2}{n_l} \implies \text{standard deviation (std)} = \sqrt{\frac{2}{n_l}}$$

The initialization according to [1] is a zero-mean Gaussian distribution whose standard deviation (std) is $\sqrt{2/n_l}$. The bias is initialized to zero. The initialization distribution therefore is of the form:

$$W_l \sim \mathcal{N} \left(\mathbf{0}, \sqrt{\frac{2}{n_l}} \right) \text{ and } \mathbf{b} = \mathbf{0}.$$

From Eqn. (4), we can observe that for the first layer ($l = 1$), the variance of weights is given by $n_l Var[w_l] = 1$ because there is no ReLU applied on the input signal. However, the factor of a single layer does not make the overall product exponentially large or small and as such we adopt Eqn. (7) in the first layer for simplicity.

Backward Propagation Case

For back-propagation, the gradient of the conv-layer is computed by:

$$\Delta \mathbf{x}_l = W_l \Delta \mathbf{y}_l. \quad (8)$$

Notation

1. $\Delta \mathbf{x}_l$ is the gradient $\frac{\partial \epsilon}{\partial \mathbf{x}}$
2. $\Delta \mathbf{y}_l$ is the gradient $\frac{\partial \epsilon}{\partial \mathbf{y}}$
3. $\Delta \mathbf{y}_l$ is represented by k by k pixels in d channels and is thus reshaped into $k^2 d$ by 1 vector i.e $\Delta \mathbf{y}_l = k^2 d \times 1$.
4. \hat{n} is given by $k^2 d$ also note that $\hat{n} \neq n$.

5. \hat{W} is a c -by- n matrix where filters are arranged in the back-propagation way. Also \hat{W} and W can be reshaped from each other.
6. $\Delta \mathbf{x}$ is a c -by-1 vector representing the gradient at a pixel of this layer.

Assumptions made here include:

- Assume that w_l and Δy_l are independent of each other then Δx_l has zero mean for all l , when w_l is initialized by a symmetric distribution around zero.
- Assume that $f'(y_l)$ and Δx_{l+1} are independent of each other

In back-propagation we have $\Delta y_l = f'(y_l) \Delta x_{l+1}$ where f' is the derivative of f . For the ReLU case $f'(y_l)$ is either zero or one with their probabilities being equal i.e $\Pr(0) = \frac{1}{2}$ and $\Pr(1) = \frac{1}{2}$.

Lets build on from some definition here; for a discrete case, the expected value of a discrete random variable, X , is found by multiplying each X -value by its probability and then summing over all values of the random variable. That is, if X is discrete,

$$E[X] = \sum_{\text{all } x} xp(x)$$

The expectation of $f'(y_l)$ then:

$$E[f'(y_l)] = (0)\frac{1}{2} + (1)\frac{1}{2} = \frac{1}{2} \quad (9)$$

With the independence of $f'(y_l)$ and Δx_{l+1} , we can show that: The expectation of $f'(y_l)$ then:

$$E[\Delta y_l] = E[f'(y_l) \Delta x_{l+1}] = E[f'(y_l)] E[\Delta x_{l+1}] \quad (10)$$

Substituting results in Eqn. (9) into Eqn. (10) we obtain:

$$E[\Delta y_l] = \frac{1}{2} E[\Delta x_{l+1}] = 0 \quad (11)$$

In Eqn. (10) Δx_l has zero mean for all l which gives us the result zero. With this we can show that $E[(\Delta y_l)^2] = \text{Var}[\Delta y_l]$ using the formula of variance as follows:

$$\begin{aligned} \text{Var}[\Delta y_l] &= E[(\Delta y_l)^2] - [E[\Delta y_l]]^2 \\ &= E[(\Delta y_l)^2] - 0 \\ &= E[(\Delta y_l)^2] \end{aligned}$$

Again with the assumption that Δx_l has zero mean for all l , we show can that the variance of product of two independent variables $f'(y_l)$ and Δx_{l+1} to be

$$\begin{aligned}
Var[\Delta y_l] &= Var[f'(y_l)\Delta x_{l+1}] \\
&= E[(f'(y_l))^2]E[(\Delta x_{l+1})^2] - [E[f'(y_l)]]^2[E[\Delta x_{l+1}]]^2 \\
&= E[(f'(y_l))^2]E[(\Delta x_{l+1})^2] - 0 \\
&= E[(f'(y_l))^2]E[(\Delta x_{l+1})^2]
\end{aligned} \tag{12}$$

From the values of $f'(y_l) \in \{0, 1\}$ we can observe that $1^2 = 1$ and $0^2 = 0$ meaning $f'(y_l) = [f'(y_l)]^2$.

This means that $E[f'(y_l)] = E[(f'(y_l))^2] = \frac{1}{2}$. Using this result in Eqn. (12) we obtain:

$$\begin{aligned}
Var[\Delta y_l] &= E[(f'(y_l))^2]E[(\Delta x_{l+1})^2] \\
&= \frac{1}{2}E[(\Delta x_{l+1})^2]
\end{aligned} \tag{13}$$

Using the formula for variance and yet again the assumption that Δx_l has zero mean for all l , we show can that $Var[\Delta x_{l+1}] = E[(\Delta x_{l+1})^2]$:

$$\begin{aligned}
Var[\Delta x_{l+1}] &= E[(\Delta x_{l+1})^2] - [E[\Delta x_{l+1}]]^2 \\
&= E[(\Delta x_{l+1})^2] - 0 \\
&= E[(\Delta x_{l+1})^2]
\end{aligned} \tag{14}$$

Substituting this result in Eqn. (13) we obtain:

$$E[(\Delta y_l)^2] = Var[\Delta y_l] = \frac{1}{2}E[(\Delta x_{l+1})^2] \tag{15}$$

The variance of Eqn. (8) can be shown to be:

$$\begin{aligned}
Var[\Delta x_{l+1}] &= \hat{n}Var[w_l]Var[\Delta y_l] \\
&= \frac{1}{2}\hat{n}Var[w_l]Var[\Delta x_{l+1}]
\end{aligned} \tag{16}$$

The scalar $1/2$ in both Eqn. (16) and Eqn. (3) is the result of ReLU, though the derivations are different. With L layers put together, we have:

$$Var[\Delta x_2] = Var[\Delta x_{L+1}] \left(\prod_{l=2}^L \frac{1}{2}\hat{n}_l Var[w_l] \right) \tag{17}$$

Considering a sufficient condition that the gradient is not exponentially large/small:

$$\frac{1}{2}\hat{n}_l Var[w_l] = 1, \quad \forall l \tag{18}$$

The only difference between Eqn. (18) and Eqn. (7) is that $\hat{n} = k_l^2 d_l$ while $n = k_l^2 c_l = k_l^2 d_{l-1}$. Eqn. (18) results in a zero-mean Gaussian distribution whose standard deviation (std) is $\sqrt{2/\hat{n}_l}$. The initialization distribution therefore is of the form:

$$w_l \sim \mathcal{N}\left(0, \sqrt{\frac{2}{\hat{n}_l}}\right)$$

For the layer ($l = 1$), we need not compute Δx because it represents the image domain. We adopt Eqn. (18) for the first layer for the same reason as the forward propagation case - the factor of a single layer does not make the overall product exponentially large or small.

It is noted that use of either Eqn. (18) or Eqn. (4) alone is sufficient. For example, if we use Eqn.(18), then in Eqn. (17) the product $\prod_{l=2}^L \frac{1}{2} \hat{n}_l \text{Var}[w_l] = 1$, and in Eqn. (4) the product $\prod_{l=2}^L \frac{1}{2} n_l \text{Var}[w_l] = \prod_{l=2}^L n_l / \hat{n} = c_2 / d_L$, which is not a diminishing number in common network designs. This means that if the initialization properly scales the backward signal, then this is also the case for the forward signal; and vice versa.

For initialization in the PReLU case, it is easy to show that Eqn. (4) becomes:

$$\frac{1}{2}(1 + a^2)n_l \text{Var}[w_l] = 1, \quad (19)$$

Where a is the initialized value of the coefficients. If $a = 0$, it becomes the ReLU case; if $a = 1$, it becomes the linear case; same as [2]. Similarly, Eqn. (14) becomes:

$$\frac{1}{2}(1 + a^2)\hat{n}_l \text{Var}[w_l] = 1, \quad (20)$$

Applications

The initialization routines derived here, more famously known as “**Kaiming Initialization**” have been successfully applied in various deep learning libraries. Below we shall look at [Keras](#) a minimalist, highly modular neural networks library, written in Python and capable of running on top of either TensorFlow or Theano.

The initialization routine here is named “`he_`” following the name of one of the authors Kaiming He [1]. In the code snippet below, **`he_normal`** is the implementation of initialization based on Gaussian distribution while **`he_uniform`** is the equivalent implementation of initialization based on Uniform distribution

```

def get_fans(shape):
    fan_in = shape[0] if len(shape) == 2 else np.prod(shape[1:])
    fan_out = shape[1] if len(shape) == 2 else shape[0]
    return fan_in, fan_out

def he_normal(shape, name=None):
    ''' Reference: He et al., http://arxiv.org/abs/1502.0
1852
    '''
    fan_in, fan_out = get_fans(shape)
    s = np.sqrt(2. / fan_in)
    return normal(shape, s, name=name)

def he_uniform(shape, name=None):
    fan_in, fan_out = get_fans(shape)
    s = np.sqrt(6. / fan_in)
    return uniform(shape, s, name=name)

```

References

1. He, Kaiming, et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." Proceedings of the IEEE International Conference on Computer Vision. 2015. [\[pdf\]](#)
2. Glorot Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." Aistats. Vol. 9. 2010. [\[pdf\]](#)
3. A. L. Maas, A. Y. Hannun, and A. Y. Ng. "Rectifier nonlinearities improve neural network acoustic models." In ICML, 2013. [\[pdf\]](#)
4. Xu, Bing, et al. "Empirical Evaluation of Rectified Activations in Convolution Network." [\[pdf\]](#)
5. Glorot, Xavier, Antoine Bordes, and Yoshua Bengio. "Deep Sparse Rectifier Neural Networks." Aistats. Vol. 15. No. 106. 2011. [\[pdf\]](#)

Related Posts

Initialization Of Feedfoward Networks 27 Jul 2016

Initialization Of Deep Feedfoward Networks 01 Aug 2016

0 Comments

Deep Grid

 Login ▾

♥ Recommend

↗ Share

Sort by Best ▾



Start the discussion...

Be the first to comment.

 Subscribe  Add Disqus to your site Add Disqus Add  Privacy