DeepLearning.AI | AI Notes

Initializing neural networks

Initialization can have a significant impact on convergence in training deep neural networks. Simple initialization schemes have been found to accelerate training, but they require some care to avoid common pitfalls. In this post, we'll explain how to initialize neural network parameters effectively.

TABLE OF CONTENTS

I The importance of effective initialization

To build a machine learning algorithm, usually you'd define an architecture (e.g. Logistic regression, Support Vector Machine, Neural Network) and train it to learn parameters. Here is a common training process for neural networks:

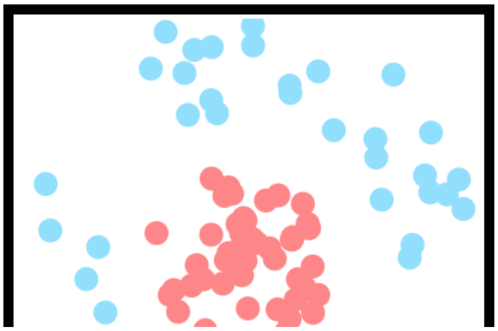
1. Initialize the parameters
2. Choose an *optimization algorithm*
3. Repeat these steps:
    1. Forward propagate an input
    2. Compute the cost function
    3. Compute the gradients of the cost with respect to parameters using backpropagation
    4. Update each parameter using the gradients, according to the optimization algorithm

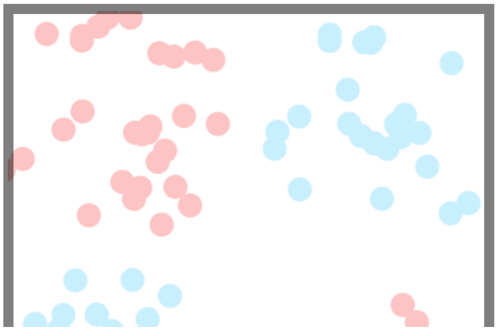
Then, given a new data point, you can use the model to predict its class.

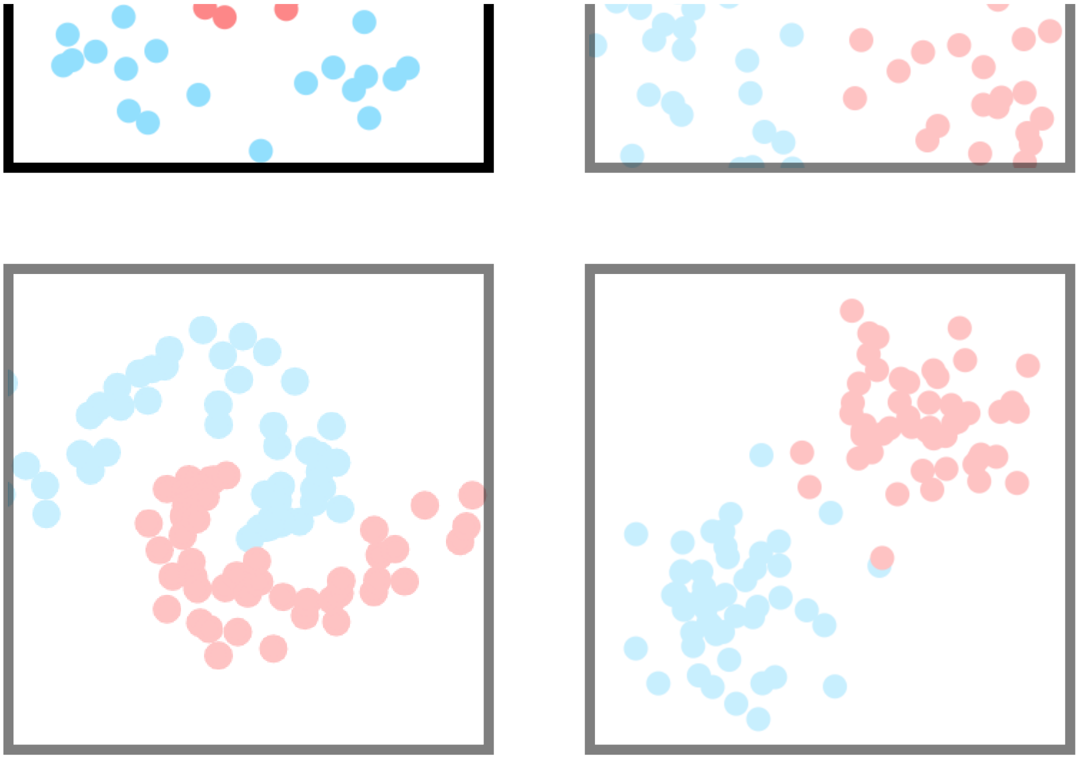
The initialization step can be critical to the model's ultimate performance, and it requires the right method. To illustrate this, consider the three-layer neural network below. You can try initializing this network with different methods and observe the impact on the learning.

1. Choose input dataset

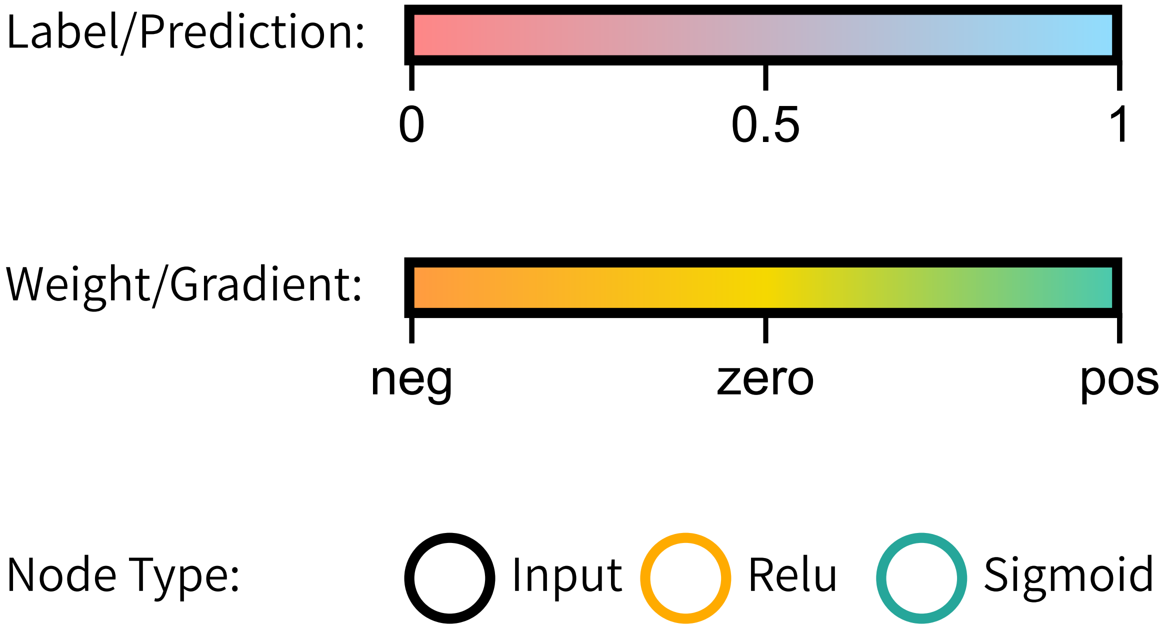
Select a training dataset.







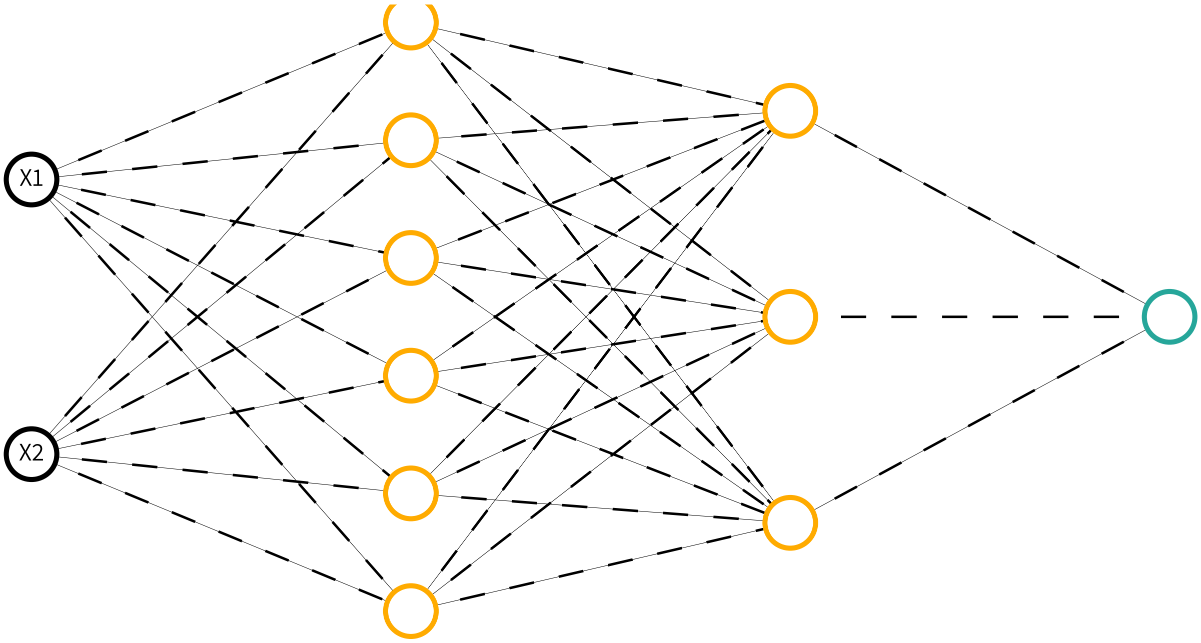
This legend details the color scheme for labels, and the values of the weights/gradients.



2. Choose initialization method

Select an initialization method for the values of your neural network parameters[1].

○ Zero ○ Too small ○ Appropriate ○ Too large



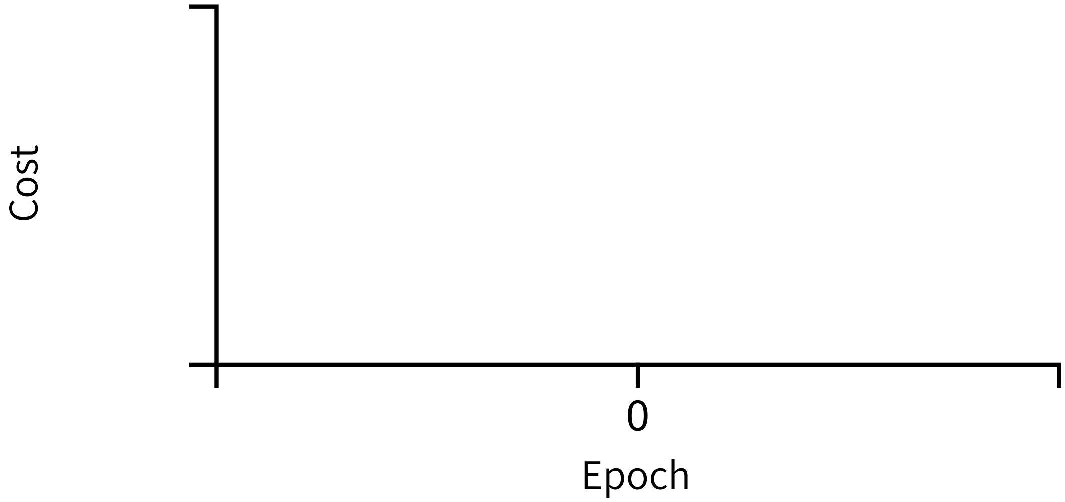
Select whether to visualize the weights or gradients of the network above.

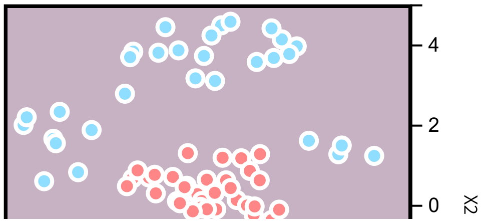
○ Weight ○ Gradient

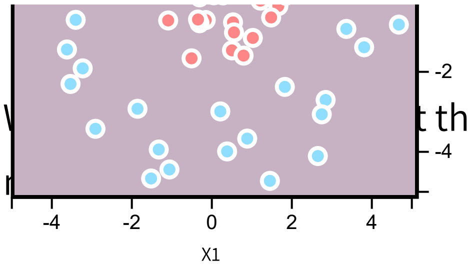
3. Train the network.

Observe the cost function and the decision boundary.

↻ ▶ ⏩







t the gradients and weights when the initialization

> *Initializing all the weights with zeros leads the neurons to learn the same features during training.*

In fact, any constant initialization scheme will perform very poorly. Consider a *neural network* with two hidden units, and assume we initialize all the biases to 0 and the weights with some constant α. If we forward propagate an input (x_1, x_2) in this network, the output of both hidden units will be $relu(\alpha x_1 + \alpha x_2)$. Thus, both hidden units will have identical influence on the cost, which will lead to identical gradients. Thus, both neurons will evolve symmetrically throughout training, effectively preventing different neurons from learning different things.

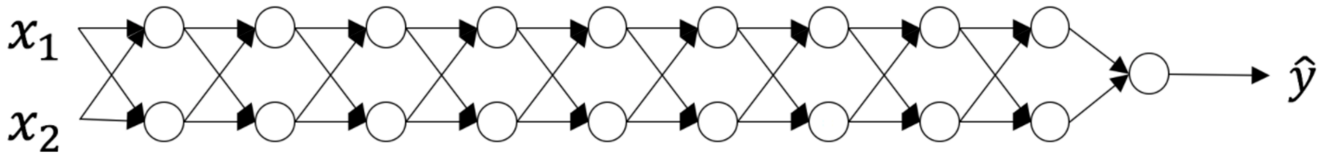
What do you notice about the cost plot when you initialize weights with values too small or too large?

> *Despite breaking the symmetry, initializing the weights with values (i) too small or (ii) too large leads respectively to (i) slow learning or (ii) divergence.*

Choosing proper values for initialization is necessary for efficient training. We will investigate this further in the next section.

II The problem of exploding or vanishing gradients

Consider this 9-layer neural network.



At every iteration of the optimization loop (forward, cost, backward, update), we observe that backpropagated gradients are either amplified or minimized as you move from the output layer towards the input layer. This result makes sense if you consider the following example.

Assume all the activation functions are linear (identity function). Then the output activation is:

$$\hat{y} = a^{[L]} = W^{[L]} W^{[L-1]} W^{[L-2]} \dots W^{[3]} W^{[2]} W^{[1]} x$$

where $L = 10$ and $W^{[1]}, W^{[2]}, \dots, W^{[L-1]}$ are all matrices of size $(2, 2)$ because layers $[1]$ to $[L-1]$ have 2 neurons and receive 2 inputs. With this in mind, and for illustrative purposes, if we assume $W^{[1]} = W^{[2]} = \dots = W^{[L-1]} = W$ the output prediction is $\hat{y} = W^{[L]} W^{L-1} x$ (where W^{L-1} takes the matrix W to the power of $L-1$, while $W^{[L]}$ denotes the L^{th} matrix).

What would be the outcome of initialization values that were too small, too large or appropriate?

Case 1: A too-large initialization leads to exploding gradients

Consider the case where every weight is initialized slightly larger than the identity matrix.

$$W^{[1]} = W^{[2]} = \dots = W^{[L-1]} = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}$$

This simplifies to $\hat{y} = W^{[L]} 1.5^{L-1} x$, and the values of $a^{[l]}$ increase exponentially with l. When these activations are used in backward propagation, this leads to the exploding gradient problem. That is, the gradients of the cost with the respect to the parameters are too big. This leads the cost to oscillate around its minimum value.

Case 2: A too-small initialization leads to vanishing gradients

Case 2: A too-small initialization leads to vanishing gradients

Similarly, consider the case where every weight is initialized slightly smaller than the identity matrix.

$$W^{[1]} = W^{[2]} = \cdots = W^{[L-1]} = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}$$

This simplifies to $\hat{y} = W^{[L]} 0.5^{L-1} x$, and the values of the activation $a^{[l]}$ decrease exponentially with l. When these activations are used in backward propagation, this leads to the vanishing gradient problem. The gradients of the cost with respect to the parameters are too small, leading to convergence of the cost before it has reached the minimum value.

All in all, initializing weights with inappropriate values will lead to divergence or a slow-down in the training of your neural network. Although we illustrated the exploding/vanishing gradient problem with simple symmetrical weight matrices, the observation generalizes to any initialization values that are too small or too large.

III How to find appropriate initialization values

To prevent the gradients of the network’s activations from vanishing or exploding, we will stick to the following rules of thumb:

1. The *mean* of the activations should be zero.
2. The *variance* of the activations should stay the same across every layer.

Under these two assumptions, the backpropagated gradient signal should not be multiplied by values too small or too large in any layer. It should travel to the input layer without exploding or vanishing.

More concretely, consider a *layer* l. Its forward propagation is:

$$a^{[l-1]} = g^{[l-1]}(z^{[l-1]})$$

$$z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$$
$$a^{[l]} = g^{[l]}(z^{[l]})$$

We would like the following to hold[2]:

$$E[a^{[l-1]}] = E[a^{[l]}]$$
$$Var(a^{[l-1]}) = Var(a^{[l]})$$

Ensuring zero-mean and maintaining the value of the variance of the input of every layer guarantees no exploding/vanishing signal, as we'll explain in a moment. This method applies both to the forward propagation (for activations) and backward propagation (for gradients of the cost with respect to activations). The recommended initialization is Xavier initialization (or one of its derived methods), for every layer l:

$$W^{[l]} \sim \mathcal{N}(\mu = 0, \sigma^2 = \frac{1}{n^{[l-1]}})$$
$$b^{[l]} = 0$$

In other words, all the weights of layer l are picked randomly from a *normal distribution* with mean $\mu = 0$ and variance $\sigma^2 = \frac{1}{n^{[l-1]}}$ where $n^{[l-1]}$ is the number of neuron in layer $l - 1$. Biases are initialized with zeros.

The visualization below illustrates the influence of the Xavier initialization on each layer's activations for a five-layer fully-connected neural network.

1. Load your dataset

Load 10,000 handwritten digits images (MNIST).

Load MNIST (0%)

2. Select an initialization method

Among the below distributions, select the one to use to initialize your parameters[3].

○ Zero ○ Uniform ○ Xavier ○ Standard Normal

3. Train the network and observe

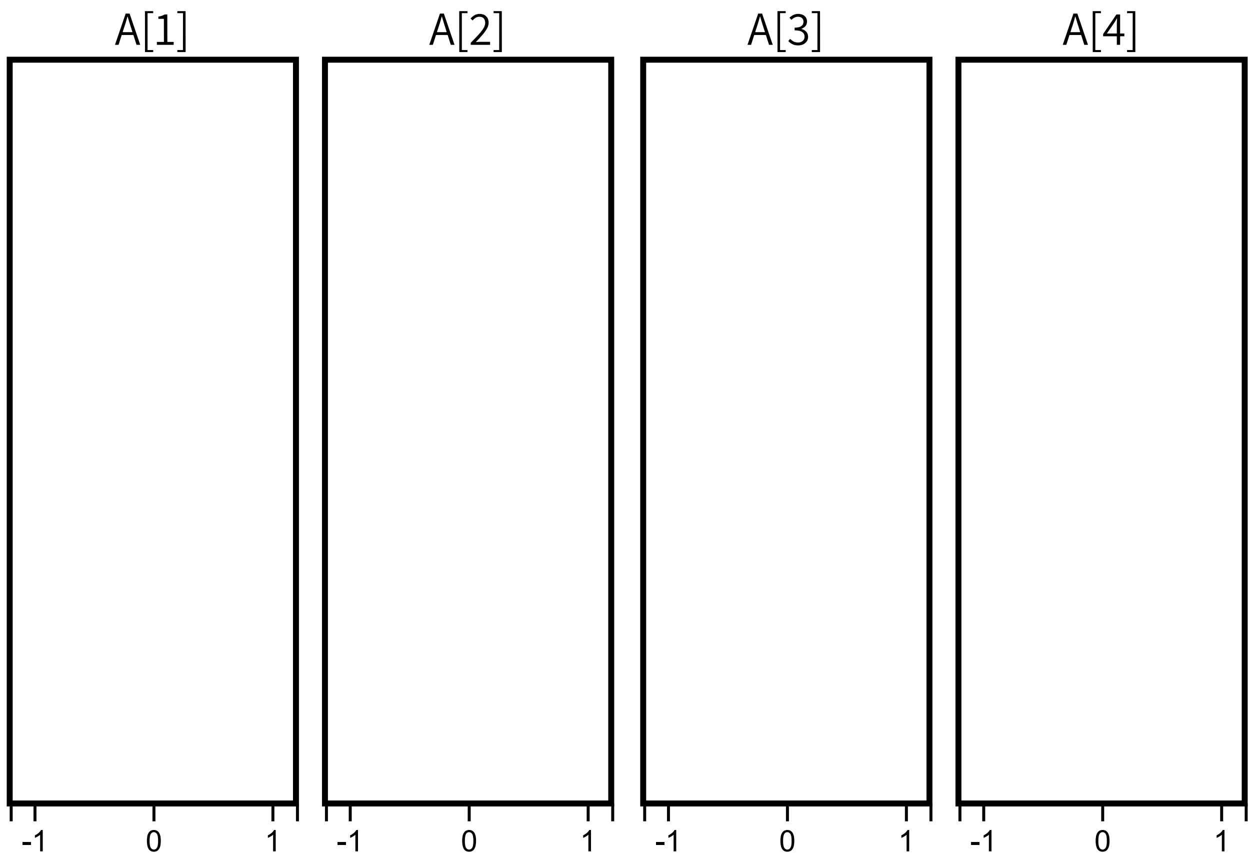
3. Train the network and observe

The grid below refers to the input images, Blue squares represent correctly classified images. Red squares represent misclassified images.

↻ ▶ ⏩

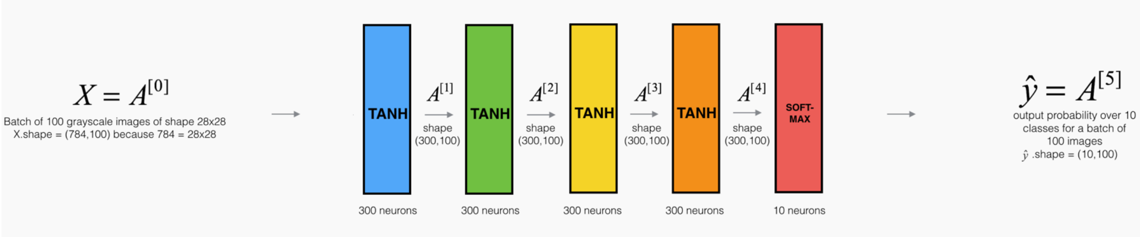
Input batch of 100 images

Batch: 0 Epoch: 0



Output predictions of 100 images

Misclassified: 0/100 Cost: 0.00



You can find the theory behind this visualization in Glorot et al. (2010). The next section presents the mathematical justification for Xavier initialization and explains more precisely why it is an effective initialization.

IV Justification for Xavier initialization

In this section, we will show that Xavier Initialization[4] keeps the variance the same across every layer. We will assume that our layer's activations are normally distributed around zero. Sometimes it helps to understand the mathematical justification to grasp the concept, but you can understand the fundamental idea without the math.

Let's work on the *layer* l described in part (III) and assume the activation function is $tanh$. The forward propagation is:

$$\begin{aligned} z^{[l]} &= W^{[l]}a^{[l-1]} + b^{[l]} \\ a^{[l]} &= tanh(z^{[l]}) \end{aligned}$$

The goal is to derive a relationship between $Var(a^{[l-1]})$ and $Var(a^{[l]})$. We will then understand how we should initialize our weights such that: $Var(a^{[l-1]}) = Var(a^{[l]})$
.

Assume we initialized our network with appropriate values and the input is normalized. Early on in the training, we are in the *linear regime* of $tanh$. Values are small enough and thus $tanh(z^{[l]}) \approx z^{[l]}$,[5] meaning that:

$$Var(a^{[l]}) = Var(z^{[l]})$$

Moreover, $z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]} = vector(z_1^{[l]}, z_2^{[l]}, \dots, z_{n^{[l]}}^{[l]})$ where $z_k^{[l]} = \sum_{j=1}^{n^{[l-1]}} w_{kj}^{[l]} a_j^{[l-1]} + b_k^{[l]}$. For simplicity, let's assume that $b^{[l]} = 0$ (it will end up being true given the choice of initialization we will choose). Thus, looking *element-wise* at the previous equation $Var(a^{[l-1]}) = Var(a^{[l]})$ now gives:

$$Var(a_k^{[l]}) = Var(z_k^{[l]}) = Var(\sum_{j=1}^{n^{[l-1]}} w_{kj}^{[l]} a_j^{[l-1]})$$

A common math trick is to extract the summation outside the variance. To do this, we must make the following three *assumptions*[6]:

1. Weights are independent and identically distributed
2. Inputs are independent and identically distributed
3. Weights and inputs are mutually independent

Thus, now we have:

$$Var(a_k^{[l]}) = Var(z_k^{[l]}) = Var(\sum_{j=1}^{n^{[l-1]}} w_{kj}^{[l]} a_j^{[l-1]}) = \sum_{j=1}^{n^{[l-1]}} Var(w_{kj}^{[l]} a_j^{[l-1]})$$

Another common math trick is to convert the variance of a product into a product of variances. Here is the *formula* for it:

$$Var(XY) = E[X]^2 Var(Y) + Var(X)E[Y]^2 + Var(X)Var(Y)$$

Using this formula with $X = w_{kj}^{[l]}$ and $Y = a_j^{[l-1]}$, we get:

$$Var(w_{kj}^{[l]} a_j^{[l-1]}) = E[w_{kj}^{[l]}]^2 Var(a_j^{[l-1]}) + Var(w_{kj}^{[l]})E[a_j^{[l-1]}]^2 + Var(w_{kj}^{[l]})Var(a_j^{[l-1]})$$

We're almost done! The first assumption leads to $E[w_{kj}^{[l]}]^2 = 0$ and the second assumption leads to $E[a_j^{[l-1]}]^2 = 0$ because weights are initialized with zero mean, and inputs are normalized. Thus:

$$Var(z_k^{[l]}) = \sum_{j=1}^{n^{[l-1]}} Var(w_{kj}^{[l]})Var(a_j^{[l-1]}) = \sum_{j=1}^{n^{[l-1]}} Var(W^{[l]})Var(a^{[l-1]}) = n^{[l-1]}Var(W^{[l]})Var(a^{[l-1]})$$

The equality above results from our first assumption stating that:

$$Var(w_{kj}^{[l]}) = Var(w_{11}^{[l]}) = Var(w_{12}^{[l]}) = \cdots = Var(W^{[l]})$$

Similarly the second assumption leads to:

$$Var(a_j^{[l-1]}) = Var(a_1^{[l-1]}) = Var(a_2^{[l-1]}) = \cdots = Var(a^{[l-1]})$$

With the same idea:

$$Var(z^{[l]}) = Var(z_k^{[l]})$$

Wrapping up everything, we have:

$$Var(a^{[l]}) = n^{[l-1]}Var(W^{[l]})Var(a^{[l-1]})$$

Voilà! If we want the variance to stay the same across layers ($Var(a^{[l]}) = Var(a^{[l-1]})$), we need $Var(W^{[l]}) = \frac{1}{n^{[l-1]}}$. This justifies the choice of variance for Xavier initialization.

Notice that in the previous steps we did not choose a specific layer l. Thus, we have shown that this expression holds for every layer of our network. Let L be the output layer of our network. Using this expression at every layer, we can link the output layer's variance to the input layer's variance:

$$\begin{aligned} Var(a^{[L]}) &= n^{[L-1]}Var(W^{[L]})Var(a^{[L-1]}) \\ &= n^{[L-1]}Var(W^{[L]})n^{[L-2]}Var(W^{[L-1]})Var(a^{[L-2]}) \\ &= \ldots \\ &= \left[\prod_{l=1}^{L} n^{[l-1]}Var(W^{[l]})\right] Var(x) \end{aligned}$$

Depending on how we initialize our weights, the relationship between the variance of our output and input will vary dramatically. Notice the following three cases.

$$n^{[l-1]}Var(W^{[l]}) \begin{cases} < 1 & \implies \text{Vanishing Signal} \\ = 1 & \implies Var(a^{[L]}) = Var(x) \\ > 1 & \implies \text{Exploding Signal} \end{cases}$$

Thus, in order to avoid the vanishing or exploding of the forward propagated signal, we must set $n^{[l-1]}Var(W^{[l]}) = 1$ by initializing $Var(W^{[l]}) = \frac{1}{n^{[l-1]}}$.

Throughout the justification, we worked on activations computed during the forward propagation. The same result can be derived for the backpropagated gradients. Doing so, you will see that in order to avoid the vanishing or exploding gradient problem, we must set $n^{[l]}Var(W^{[l]}) = 1$ by initializing $Var(W^{[l]}) = \frac{1}{n^{[l]}}$.

Conclusion

In practice, Machine Learning Engineers using Xavier initialization would either initialize the weights as $\mathcal{N}(0, \frac{1}{n^{[l-1]}})$ or as $\mathcal{N}(0, \frac{2}{n^{[l-1]}+n^{[l]}})$. The variance term of the latter distribution is the harmonic mean of $\frac{1}{n^{[l-1]}}$ and $\frac{1}{n^{[l]}}$.

This is a theoretical justification for Xavier initialization. Xavier initialization works with tanh activations. Myriad other initialization methods exist. If you are using ReLU, for example, a common initialization is He initialization (He et al., Delving Deep into Rectifiers), in which the weights are initialized by multiplying by 2 the variance of the Xavier initialization. While the justification for this initialization is slightly more complicated, it follows the same thought process as the one for tanh.

Learn more about how to effectively initialize parameters in Course 2 of the Deep Learning Specialization

Enroll now

AUTHORS

1. Kian Katanforoosh - Written content and structure.
2. Daniel Kunin - Visualizations (created using D3.js and TensorFlow.js).

ACKNOWLEDGMENTS

1. The template for the article was designed by Jingru Guo and inspired by Distill.
2. The first visualization adapted code from Mike Bostock's visualization of the Goldstein-Price function.
3. The banner visualization adapted code from deeplearn.js's implementation of a CPPN.

FOOTNOTES

1. All bias parameters are initialized to zero and weight parameters are drawn from a normal distribution with zero mean and selected variance.

2. Under the hypothesis that all entries of the weight matrix $W^{[l]}$ are picked from the same distribution, $Var(w_{11}) = Var(w_{12}) = \cdots = Var(w_{n^{[l]}n^{[l-1]}})$. Thus, $Var(W^{[l]})$ indicates the variance of any entry of $W^{[l]}$ (they're all the same!). Similarly, we will denote $Var(x)$ (resp. $Var(a^{[l]})$) the variance of any entry of x (resp. $a^{[l]}$). It is a fair approximation to consider that every pixel of a "real-world image" x is distributed according to the same distribution.

3. All bias parameters are initialized to zero and weight parameters are drawn from either "Zero" distribution ($w_{ij} = 0$), "Uniform" distribution ($w_{ij} \sim U(\frac{-1}{\sqrt{n^{[l-1]}}}, \frac{1}{\sqrt{n^{[l-1]}}})$), "Xavier" distribution ($w_{ij} \sim N(0, \frac{1}{\sqrt{n^{[l-1]}}})$), or "Standard Normal" distribution ($w_{ij} \sim N(0, 1)$).

4. Concretely it means we pick every weight randomly and independently from a normal distribution centered in $\mu = 0$ and with variance $\sigma^2 = \frac{1}{n^{[l-1]}}$.

5. We assume that $W^{[l]}$ is initialized with small values and $b^{[l]}$ is initialized with zeros. Hence, $Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$ is small and we are in the linear regime of $tanh$. Remember the slope of $tanh$ around zero is one, thus $tanh(Z^{[l]}) \approx Z^{[l]}$.

6. The first assumption will end up being true given our initialization scheme (we pick weights randomly according to a normal distribution centered at zero). The second assumption is not always true. For instance in images, inputs are pixel values, and pixel values in the same region are highly correlated with each other. On average, it’s more likely that a green pixel is surrounded by green pixels than by any other pixel color, because this pixel might be representing a grass field, or a green object. Although it’s not always true, we assume that inputs are distributed identically (let’s say from a normal distribution centered at zero.) The third assumption is generally true at initialization, given that our initialization scheme makes our weights independent and identically distributed (i.i.d.).

REFERENCE

To reference this article in an academic context, please cite this work as:

```
Katanforoosh & Kunin, "Initializing neural networks", deeplearning.ai, 2019.
```