1. Is it okay to initialize all the weights to the same value as long as that value is selected randomly using He initialization?

No matter what was the input - if all weights are the same, all units in hidden layer will be the same too. This is the main issue with symmetry and reason why you should initialize weights randomly (or, at least, with different values). Note, that this issue affects all architectures that use each-to-each connections.

Building even a simple neural network can be a confusing task and tuning it to get the better result is an extremely tedious task. The most common problem with Deep Neural Networks is Vanishing and Exploding gradient descent. To solve these issues, one solution could be to initialize the parameters carefully. In this article, we will discuss Weight initialization techniques.

This article has been written under the assumption that you have a basic understanding of neural networks, weights, biases, activation functions, forward and backward propagation, etc.

Table of Contents

 Basics and notations of neural networks

 Steps of training a neural network

 Why weight initialization?

 Different Weight initialization techniques

 Best practices of weight initialization

 Conclusion

Basics and Notations

Consider a neural network having an l layer, which has l-1 hidden layers and 1 output layer. Then, the parameters i.e, weights and biases of the layer l are represented as,

In addition to weights and biases, some more intermediate variables are also computed during the training process,

Steps of Training a Neural Network

Training a neural network consists of the following basic steps:

Step-1: Initialization of Neural Network: Initialize weights and biases.

Step-2: Forward propagation: Using the given input X, weights W, and biases b, for every layer we compute a linear combination of inputs and weights (Z)and then apply activation function to linear combination (A). At the final layer, we compute f(A(l-1)) which could be a sigmoid (for binary classification problem), softmax (for multi-class classification problem), and this gives the prediction y\_hat.

Step-3: Compute the loss function: The loss function includes both the actual label y and predicted label y\_hat in its expression. It shows how far our predictions from the actual target, and our main objective is to minimize the loss function.

Step-4: Backward Propagation: In backpropagation, we find the gradients of the loss function, which is a function of y and y\_hat, and gradients wrt A, W, and b called dA, dW, and db. By using these gradients, we update the values of the parameters from the last layer to the first layer.

Step-5: Repeat steps 2–4 for n epochs till we observe that the loss function is minimized, without overfitting the train data.

2.Is it okay to initialize the bias terms to 0?

It is possible and common to initialize the biases to be zero, since the asymmetry breaking is provided by the small random numbers in the weights.

3.Name three advantages of the ELU activation function over ReLU.

ELU is very similiar to RELU except negative inputs. They are both in identity function form for non-negative inputs.  
...  
ELU

ELU becomes smooth slowly until its output equal to -α whereas RELU sharply smoothes.

ELU is a strong alternative to ReLU.

Unlike to ReLU, ELU can produce negative outputs.

4.In which cases would you want to use each of the following activation functions: ELU, leaky ReLU (and its variants), ReLU, tanh, logistic, and softmax?

[Linear](https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html#toc-entry-1)

A straight line function where activation is proportional to input ( which is the weighted sum from neuron ).

|  |  |
| --- | --- |
| Function | Derivative |
| R(z,m)={z∗m}R(z,m)={z∗m} | R′(z,m)={m}R′(z,m)={m} |
|  |  |
| def linear(z,m):  return m\*z | def linear\_prime(z,m):  return m |

Pros

It gives a range of activations, so it is not binary activation.

We can definitely connect a few neurons together and if more than 1 fires, we could take the max ( or softmax) and decide based on that.

Cons

For this function, derivative is a constant. That means, the gradient has no relationship with X.

It is a constant gradient and the descent is going to be on constant gradient.

If there is an error in prediction, the changes made by back propagation is constant and not depending on the change in input delta(x) !

[ELU](https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html#toc-entry-2)

Exponential Linear Unit or its widely known name ELU is a function that tend to converge cost to zero faster and produce more accurate results. Different to other activation functions, ELU has a extra alpha constant which should be positive number.

ELU is very similiar to RELU except negative inputs. They are both in identity function form for non-negative inputs. On the other hand, ELU becomes smooth slowly until its output equal to -α whereas RELU sharply smoothes.

|  |  |
| --- | --- |
| Function | Derivative |
| R(z)={zα.(ez–1)z>0z<=0}R(z)={zz>0α.(ez–1)z<=0} | R′(z)={1α.ezz>0z<0}R′(z)={1z>0α.ezz<0} |
|  |  |
| def elu(z,alpha):  return z if z >= 0 else alpha\*(e^z -1) | def elu\_prime(z,alpha):  return 1 if z > 0 else alpha\*np.exp(z) |

Pros

ELU becomes smooth slowly until its output equal to -α whereas RELU sharply smoothes.

ELU is a strong alternative to ReLU.

Unlike to ReLU, ELU can produce negative outputs.

Cons

For x > 0, it can blow up the activation with the output range of [0, inf].

[ReLU](https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html#toc-entry-3)

A recent invention which stands for Rectified Linear Units. The formula is deceptively simple: max(0,z)max(0,z). Despite its name and appearance, it’s not linear and provides the same benefits as Sigmoid (i.e. the ability to learn nonlinear functions), but with better performance.

|  |  |
| --- | --- |
| Function | Derivative |
| R(z)={z0z>0z<=0}R(z)={zz>00z<=0} | R′(z)={10z>0z<0}R′(z)={1z>00z<0} |
|  |  |
| def relu(z):  return max(0, z) | def relu\_prime(z):  return 1 if z > 0 else 0 |

Pros

It avoids and rectifies vanishing gradient problem.

ReLu is less computationally expensive than tanh and sigmoid because it involves simpler mathematical operations.

Cons

One of its limitations is that it should only be used within hidden layers of a neural network model.

Some gradients can be fragile during training and can die. It can cause a weight update which will makes it never activate on any data point again. In other words, ReLu can result in dead neurons.

In another words, For activations in the region (x<0) of ReLu, gradient will be 0 because of which the weights will not get adjusted during descent. That means, those neurons which go into that state will stop responding to variations in error/ input (simply because gradient is 0, nothing changes). This is called the dying ReLu problem.

The range of ReLu is [0,∞)[0,∞). This means it can blow up the activation.

Further reading

[Deep Sparse Rectifier Neural Networks](http://proceedings.mlr.press/v15/glorot11a/glorot11a.pdf) Glorot et al., (2011)

[Yes You Should Understand Backprop](https://medium.com/@karpathy/yes-you-should-understand-backprop-e2f06eab496b), Karpathy (2016)

[LeakyReLU](https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html#toc-entry-4)

LeakyRelu is a variant of ReLU. Instead of being 0 when z<0z<0, a leaky ReLU allows a small, non-zero, constant gradient αα (Normally, α=0.01α=0.01). However, the consistency of the benefit across tasks is presently unclear. [[1]](https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html#footnote-1)

|  |  |
| --- | --- |
| Function | Derivative |
| R(z)={zαzz>0z<=0}R(z)={zz>0αzz<=0} | R′(z)={1αz>0z<0}R′(z)={1z>0αz<0} |
|  |  |
| def leakyrelu(z, alpha):  return max(alpha \* z, z) | def leakyrelu\_prime(z, alpha):  return 1 if z > 0 else alpha |

Pros

Leaky ReLUs are one attempt to fix the “dying ReLU” problem by having a small negative slope (of 0.01, or so).

Cons

As it possess linearity, it can’t be used for the complex Classification. It lags behind the Sigmoid and Tanh for some of the use cases.

Further reading

[Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification](https://arxiv.org/pdf/1502.01852.pdf), Kaiming He et al. (2015)

[Sigmoid](https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html#toc-entry-5)

Sigmoid takes a real value as input and outputs another value between 0 and 1. It’s easy to work with and has all the nice properties of activation functions: it’s non-linear, continuously differentiable, monotonic, and has a fixed output range.

|  |  |
| --- | --- |
| Function | Derivative |
| S(z)=11+e−zS(z)=11+e−z | S′(z)=S(z)⋅(1−S(z))S′(z)=S(z)⋅(1−S(z)) |
|  |  |
| def sigmoid(z):  return 1.0 / (1 + np.exp(-z)) | def sigmoid\_prime(z):  return sigmoid(z) \* (1-sigmoid(z)) |

Pros

It is nonlinear in nature. Combinations of this function are also nonlinear!

It will give an analog activation unlike step function.

It has a smooth gradient too.

It’s good for a classifier.

The output of the activation function is always going to be in range (0,1) compared to (-inf, inf) of linear function. So we have our activations bound in a range. Nice, it won’t blow up the activations then.

Cons

Towards either end of the sigmoid function, the Y values tend to respond very less to changes in X.

It gives rise to a problem of “vanishing gradients”.

Its output isn’t zero centered. It makes the gradient updates go too far in different directions. 0 < output < 1, and it makes optimization harder.

Sigmoids saturate and kill gradients.

The network refuses to learn further or is drastically slow ( depending on use case and until gradient /computation gets hit by floating point value limits ).

5.What may happen if you set the momentum hyperparameter too close to 1 (e.g., 0.99999) when using a MomentumOptimizer?

If you set the momentum hyperparameter too close to 1 (e.g., 0.99999) when using an SGD optimizer, then the algorithm will likely pick up a lot of speed, hopefully moving roughly toward the global minimum, but its momentum will carry it right past the minimum.

6. Name three ways you can produce a sparse model.

What is sparse model?

regression.

machine-learning.

lasso.

regularization.

ridge-regression.

7.Does dropout slow down training? Does it slow down inference (i.e., making predictions on new instances)?

More importantly, when using MC Dropout you generally want to run inference 10 times or more to get better predictions. This means that making predictions is slowed down by a factor of 10 or more. works by going from the output layer to the input layer, propagating the error gradient along the way.