Parameter Initialization

One of the first things that you will want do is to initialize the weight matrices and bias vectors. Their initial values for the weight matrices should not be zero since the cost reduction gradients would be the same value for each iteration (all weights associated with a particular parameters will be the same value) causing the learning algorithm to not learn. This basically makes you neural network no better than a linear model. This is also called the **problem of symmetric weights**. So, it's best to randomly choose the initial weight values that are between -1 and 1 and multiply them by a small scalar such as 0.01 to make the activation units active and be on the regions where activation functions' derivatives are not close to zero.

> **NOTE:** Bias weights can still be initialized to 0. This is because the gradients with respect to bias depend only on the linear activation of that layer, and not on the gradients of the deeper layers.

Mathematically, we want to initialize each $\Theta_{ij}^{(l)}$ weight to a random value in $\big[-\epsilon, \epsilon\big]$ typically $[-1, 1]$:

> $$-\epsilon \leq \Theta_{ij}^{(l)} \leq \epsilon$$

However, when initializing weights randomly with deep networks, you can run into two main issues: The problems of *vanishing gradients* and *exploding gradients*. During back propagation the gradients get smaller and smaller it approaches the global minimum as it should. However, you can run into the *problem of vanishing gradients* especially with *sigmoid* or *tanh* activation functions. This will prevent weights from changing their value. **ReLU overcomes the problem of vanishing gradients since the gradient is 0 for numbers ≤ 0 and 1 for positive numbers.**

When weights are large and > 0 and small activation functions like *sigmoid* are used, the change in cost $J(\Theta)$ will be quite large and result in *exploding gradients* during back propagation. **Exploding gradients may result in oscillating around the minima or even overshooting the optimum again and again causing the model to not learn.** Exploding gradients can get so large they **can also cause overflows and result in NaN**.

Best Practices to Avoid Vanishing and Exploding Gradients

Using ReLU

Consider using *ReLU* or *Leaky ReLU* for activation functions where you have deep networks prone to vanishing gradients. Leaky ReLU overcomes vanishing gradients entirely by supplying a small, non-zero gradient, α value (usually 0.01) for z values ≤ 0.

Weight Initialization with Heuristics

For deep networks using heuristics (logic) to initialize weights can help mitigate gradient issues. We create out weight values based on the types of non-linear activation functions that are used. The heuristics outlined below are better since they take the size of the network into consideration.

When Using ReLU and Leaky ReLU

This is also called *He Initialization*. For ReLU we multiply our randomly generated weights by the variance $\sqrt{\frac{k}{n}}$ where k is dependent on the activation function used and n is the size (number of activation units or neurons) in the previous layer. This makes the weights inversely proportional to the square root of the number of neurons in the previous layer. One common problem when using ReLU activation functions is that it has the common problem of *dying neurons*. Up to 50% of the neurons can die (stop learning) during training that's why the factor is multiplied by 2.

> $$W^{[l]} = W^{[l]} \sqrt{\frac{2}{size^{[l-1]}}}$$

Implementation in Python:

```
w[l] = np.random.randn(size[l], size[l - 1]) * np.sqrt(2 / size[l - 1])
```

When Using Sigmoid and Tanh

This is also called *Xavier Initialization* and is similar to *He* except $k = 1$ instead of 2.

> $$W^{[l]} = W^{[l]} \sqrt{\frac{1}{size^{[l-1]}}}$$

Other

Another common heuristic similar to *Xavier* and *He* is:

> $$W^{[l]} = W^{[l]} \sqrt{\frac{2}{size^{[l-1]} + size^{[l]}}}$$

Gradient Clipping

Gradient clipping is another method to help mitigate exploding gradients were the gradient is set to a value if it drops below a predefined threshold. For example, you can normalize the gradients when L2 regularization exceeds a certain threshold:

> $$W^{[l]} = W^{[l]} \cdot \frac{t}{\lambda \sum_{j=1}^{n} \theta_j^2}$$ when L2 regularization of $W^{[l]} > t$

Python Implementation

Below is a Python example that initializes the weight matrices W for each node W^i with a uniform distribution of random values using *Xavier Initialization*. Bias vectors are initialized to zeros. It accepts an array or list which contains the dimensions for each layer and returns a matrix and bias vector for each layer.

```
from scipy.stats import truncnorm

# We like to create random numbers with a normal (Gaussian) distribution,
# but the numbers have to be bounded which is why the use of truncnorm.
def truncated_normal(mean=0, sd=2, low=-1, upp=1):
    return truncnorm(
        (low - mean) / sd, (upp - mean) / sd, loc=mean, scale=sd)

def xavier(prev_layer_size):
    return np.sqrt(1 / prev_layer_size)

def initialize_parameters(layers_dims):
    np.random.seed(1)
```

```
    parameters = {}

    # Get the number of layers
    L = len(layers_dims)

    # For each layer initalize the weights and bias vector
    for l in range(1, L):
        parameters["W" + str(l)] = np.random.randn(
            layers_dims[l], layers_dims[l - 1]) * xavier(layers_dims[l - 1])
        parameters["b" + str(l)] = np.zeros((layers_dims[l], 1))

        assert parameters["W" + str(l)].shape == (
            layers_dims[l], layers_dims[l - 1])
        assert parameters["b" + str(l)].shape == (layers_dims[l], 1)

    return parameters
```

Sources

https://medium.com/usf-msds/deep-learning-best-practices-1-weight-initialization-14e5c0295b94

https://towardsdatascience.com/why-better-weight-initialization-is-important-in-neural-networks-ff9acf01026d

https://www.youtube.com/watch?v=OF8ocg5mgx0&list=PLLssT5z_DsK-h9vYZkQkYNWcItqhlRJLN&index=55

https://www.python-course.eu/neural_networks_with_python_numpy.php