CS 282 Deep Neural Networks **Spring 2023**

Lecture 6

Lecturer: Anant Sahai *Scribes: Mohamed Elgharbawy, Jai Sankar*

1. Initialization

1.1 Insight

In the previous lectures, we know that given a ReLU network, the elbow (corner) can be found at $c = -\frac{b}{w}$.

Let $w \sim N(0,1)$ and $b \sim N(0,1)$. The distribution of the elbow turns out to be Cauchy.

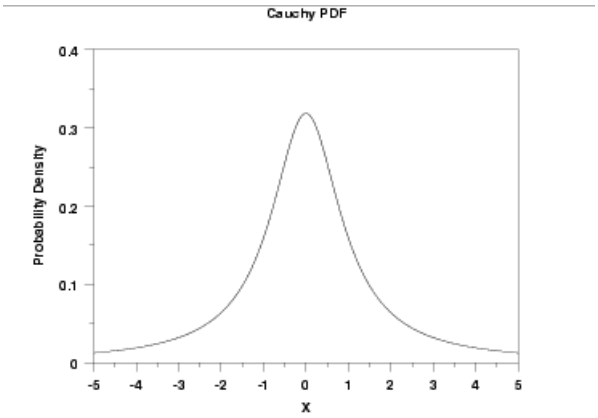

Figure 6.1: Cauchy Probability Density Function

The PDF of the corner c:

$$f(c) = \frac{1}{\pi(1+c^2)} \tag{6.1}$$

If we try to find the expectation of the corner, it turns out $E[C]$ is actually undefined due to it diverging:

$$E[C] = \int_{-\infty}^{\infty} \frac{1 \cdot c}{\pi(1+c^2)} dc \rightarrow \text{diverges} \tag{6.2}$$

Why do we care? It's a desirable quality for the input to our ReLU layers to be centered near 0 and to be close to where the "action" is, otherwise poor initialization can lead to dead ReLU nodes from the beginning. Dead ReLUs are problematic because they make all the computations with the weights and biases, but are always less than 0, leading to wasted computations. Additionally, dead ReLUs are usually unable to recover, leading to a portion of our network being unable to learn further. If we can achieve a distribution of $N(0,1)$ at initialization, the outputs of the layer will be standardized.

Idea: Can our inputs have a distribution of $N(0,1)$ at initialization?

1.2 Weight Initialization

1.2.1 Xavier Initialization

Xavier Initialization aims to initialize weights such that the variance across every layer is the same. Let d be the fan-in (number of inputs of the layer) of the target layer.

Xavier Initialization

$$w_{1,i} \overset{\text{iid}}{\sim} N\left(0, \frac{1}{d}\right) \tag{6.3}$$

Problem: This doesn't work well for ReLU!

Why? If the previous layer was ReLU, we expect half of the outputs to be 0 on average. We really only have $\frac{d}{2}$ inputs, causing our the sum of layer's variances to be $\frac{1}{2}$ instead of 1.

1.2.2 He Initialization

In order to fix the variance issue from Xavier initialization, we simply multiply our variance by 2 such that the sum of the layer's variances equals 1.

He Initialization

$$w_{1,i} \overset{\text{iid}}{\sim} N\left(0, \frac{2}{d}\right) \tag{6.4}$$

We use He Initialization when the previous layer is ReLU, in order to achieve our target $N(0,1)$ initialization distribution.

1.3 Bias Initialization

There are typically four approaches that are used to initialize biases:

1. **Treat fan-in as $d+1$ and use Xavier Initialization**
   This treats the bias as a weight in order to use a weight initialization technique, which is why we use $d+1$ instead of d for the fan-in.
2. **Just use bias $b = 0$**
   This is essentially letting our optimizer initialize the bias in the first few steps.
3. **Just use $b = 0.01$**
   Empirically, people have seen that this sometimes performs better than $b = 0$.
4. **Just use any small, random number**
   This claims that it doesn't matter what small, random number we select, since our optimizer will quickly change it anyways.

2. Optimizers

2.1 Motivation

We want to find a way to minimize the optimizer's time of convergence for covariates with relevant singular values, while having our optimizer ignore non-relevant singular values.

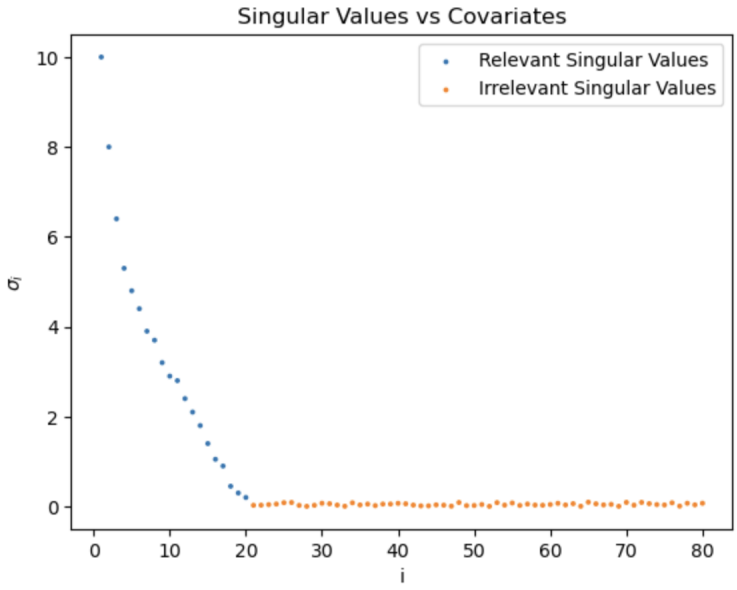

Figure 6.2: Relevant vs Irrelevant Singular Values[1]

The issue is that our singular values require different learning rates to converge quickly without oscillating. For example, our largest singular value requires a smaller learning rate so that it doesn't diverge, however, this learning rate would cause slow convergence for the smaller singular values. Thus, we want an optimization method that allows for larger learning rates for smaller singular values, while avoiding instability for larger singular values.

2.2 Momentum

If we abstract this problem even further, we would like to preserve oscillations in one direction, while preserving steadiness in another. We have a solution for this, which is low-pass filters.

Idea: To achieve quick convergence in relevant singular values, we can low-pass filter our gradients.

In essence, we want to find the simplest average of our gradient to achieve this low-pass filter. In a Low-Pass Filter, the voltage is regulated once it hits a cutoff frequency.

[1]Original image generated using Matplotlib

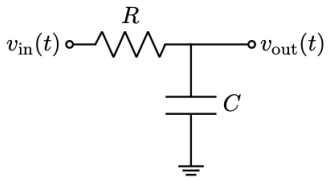

Figure 6.3: Low-Pass Filter Circuit

This low-pass filter (LPF) will regulate voltage. The discrete time first order differential-equation to achieve this would be:

Discrete-Time LPF Differential Equation

$$\begin{gathered} a_{t+1} = (1-\beta)a_t + \beta\mu_t \\ \text{Where } \mu_t \text{ is a time sequence and } \beta \in (0,1) \end{gathered} \tag{6.5}$$

This average is beneficial, as it only requires one internal state to keep to track of, and β will determine how long averaging will go for. This technique is commonly referred to as exponential smoothing.

Now we want to apply this technique to create a more stable gradient descent. Recall that the original gradient descent follows the formula:

Original Gradient Descent

$$\theta_{t+1} = \theta_t + \eta(-\nabla f_{\theta_t}) \tag{6.6}$$

In order to achieve gradient descent with momentum, we want to replace our gradient term in regular gradient descent with our low-pass filter equation applied to our gradient. Gradient descent with momentum can be considered as a "smooth gradient".

Gradient Descent with Momentum

$$\begin{aligned} \theta_{t+1} &= \theta_t + \eta(\alpha_{t+1}) \\ \alpha_{t+1} &= (1-\beta)\alpha_t + \beta(-\nabla f_{\theta_t}) \end{aligned} \tag{6.7}$$

The main difference with momentum is that gradients at previous time steps have an impact on the current gradient's value.

What we wish this lecture also had to make things clearer

A demo of the convergence of parameters for gradient descent vs gradient descent with momentum would have been nice to better understand how momentum differs from standard gradient descent conceptually.