浙江大學

Shannon Class Report

Title	**Neural Networks Overview**
Name	**He Yangshuo**
ID	**3180102687**
Tutor	**Yu Guanding**
Grade	**2018**
Major	**InformationEngineering**
College	**ISEE**
Date	**2020.2.20**

Abstract

What is a neural network and how does it successfully work? This article provides an introduction of neural networks, an overview of the optimization algorithms for training and some main theory questions regarding Deep Neural Networks. First, we start with some basic intuitions of a neural network and some detailed implementations about forward propagation and backward propagation. Second, we review the obstacle in training Deep Neural Networks and the corresponding solution including initialization and generic optimization algorithms. Third, we discuss the three main theory questions about Deep Neural Networks.

List of Contents

1 Introduction

AI is the new electricity that brings about a big transform in the society, ranging from commercial advertising, health care, auto driving, agriculture and many others. Deep Neural Networks demonstrate its impressive power in mathematical problems, computer vision, natural language processing etc. However, to most people these state-of-the-art neural networks remain to be black-boxes even they have successfully taken them into applications. Still, to bring the potential of neural networks into full play, it is inevitable to have a better command of what neural networks is doing and why neural networks work so well?

This article aims to cast light on neural networks. In Section 2, we first look into the foundation of neural networks including the representation of a neuron, the process of forward propagation and back propagation, and the basic intuition of gradient descent (GD). In Section 3, we briefly summarize the most common problems during training a neural network. Subsequently, in Section 4, we are going to introduce some effective tricks including a overview of initialization and optimization, to overcome these problems. In Section 5, we take a short look at the three main parts in neural networks theory, they cover: the power of approximation of deep neural networks, the landscape of the empirical risk and the generalization of GD.

2 Neural Network

In this article, we focus mainly on supervised learning rather than unsupervised learning. In most cases, the tasks of neural networks turn out to be regression, binary classification for instance. Generally speaking, a neural network approximate a function by minimizing the difference between the ideal one and the approximated one. In short, what neural networks do is optimization.

In this section, we start with the representation of a neural network and then further into the way neural networks solve an optimization problem.

2.1 Neuron

Neuron is an abstract representation of the smallest component of a neural network. Specifically neuron is a type of function $g : \mathbb{R}^n \to \mathbb{R}^n$, which is also called activation function including ReLU, sigmoid, softmax etc. In various neural networks such as Deep Neural Network, Convolution Neural Network (CNN) and Recurrent Neural Network (RNN), neuron takes on difference forms.

The basic model of neuron is illustrated in Figure 2.1. We denote that $x \in \mathbb{R}^n$ is the input data, $W \in \mathbb{R}^n$ and $b \in \mathbb{R}$ are the parameters in the neuron, $z \in \mathbb{R}$ is a cache given by:

$$z = W^T x + b \tag{2-1}$$

$a \in \mathbb{R}$ is the output of a neuron given by:

$$a = g(z) \tag{2-2}$$

$g : \mathbb{R}^n \to \mathbb{R}^n$ is the activation function.

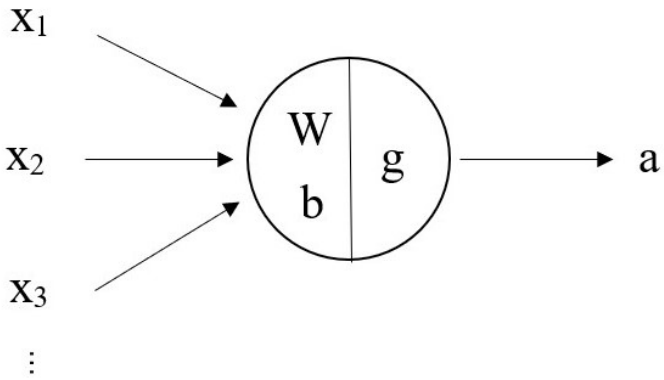


Figure 2.1: The model of neuron

The most common used activation functions are ReLU, sigmoid which are illustrated in Figure 3.2.

(a) ReLU: $g(z) = max(0,\ z)$

(b) sigmoid: $\sigma(z) = \frac{1}{1+e^{(-z)}}$

Figure 2.2: Common activation functions

In multi-classification tasks, softmax $g(z) = \frac{e^z}{\sum_{i=1}^{n} e^{z_i}},\ z \in \mathbb{R}^n$ is widely used in the last neuron of the neural network.

2.2 Architecture

Various neurons form the neural networks, and the specific formation is called the architecture of the neural networks. Most of the architecture like Deep Neural Network, CNN and RNN can be regarded as a set of layers, each layer consists of a bunch of

neurons. The output of the previous layers serve as the input of the neurons in the current layer.

First we define the following notations: l is the index of layer, L is the number of layers, $n^{[l]}$ is the number of neurons in the l^{th} layer, $W^{[l]} \in \mathbb{R}^{n^{[l]} \times n^{[l-1]}}$ and $b^{[l]} \in \mathbb{R}^{n^{[l]} \times 1}$ are the weights of neurons in the l^{th} layer, $z^{[l]} \in \mathbb{R}^{n^{[l]} \times 1}$ is the caches of neurons in the l^{th} layer, $a^{[l]} \in \mathbb{R}^{n^{[l]} \times 1}$ is the output of neurons in the l^{th} layer, $g^{[l]} : \mathbb{R}^{n^{[l]} \times 1} \to \mathbb{R}^{n^{[l]} \times 1}$ is the activation function in the l^{th} layer.

The basic model of a neural network is demonstrated in Figure 2.3, the input x can also be considered as the 0^{th} layer, which means $x = a^{[0]}$, and it is called the input layer. The last layer i.e. the L^{th} layer is called the output layer which means $\hat{y} = a^{[L]}$. The remainder are called the hidden layers.

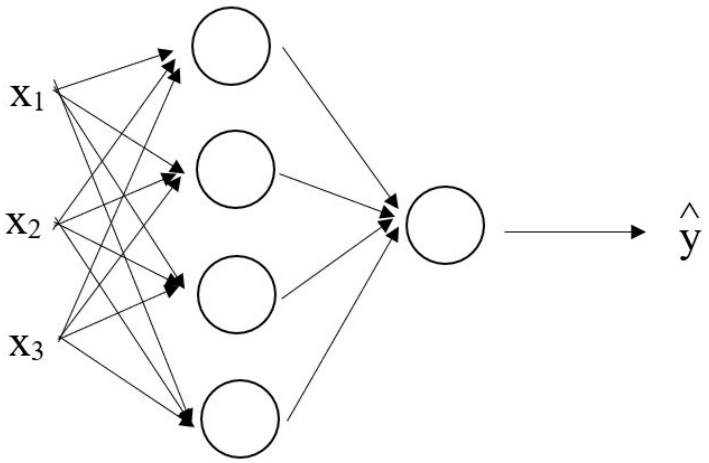


Figure 2.3: Shallow Neural Network

Shallow network is a simple type of neural network with only one hidden layer, while deep network, see Figure 2.4, has multiple hidden layers thus is much more complicated than the shallow one.

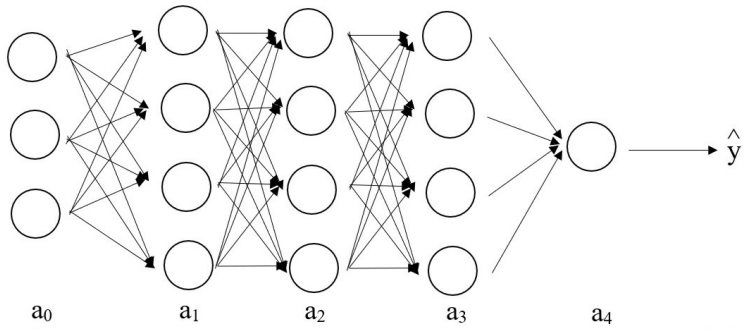


Figure 2.4: Deep Neural Network

2.3 Loss Function

The core of optimization problem is to minimize the cost. Given a set of labeled data $x,\ y,\ x \in \mathbb{R}^{n^{[0]} \times m},\ y \in \mathbb{R}^{1 \times m}$, we denote that m is the number of data. We want to pick the parameters $W,\ b$ in each neurons so that the predicted output $\hat{y}$ is close to the true value y. To measure the difference we need to calculate the distance between the predicted $\hat{y}^{(i)}$ from the i^{th} sample and $y^{(i)}$. For a certain distance metric $l(\hat{y}^{(i)},\ y^{i})$ which is also be regarded as loss function, we can write the total cost $J(\hat{y},\ y)$ of the predicted $\hat{y}$ and define the optimization problem as:

$$
\begin{aligned}
&Given\ \{x,\ y\} \\
&\arg\min_{W,\ b} J(\hat{y},\ y) \\
&J(\hat{y},\ y) = \sum_{i=1}^{m} l(\hat{y}^{(i)},\ y^{i})
\end{aligned}
\tag{2-3}
$$

As for the choice of l, the most widely used one is L_2 Norm i.e. Euclidean Distance $l(x,\ y) = ||x - y||_2^2$. Nevertheless, in difference kinds of problems, there would be a corresponding loss function that has the best performance. For instance, in binary classification problem, the popular choice is binary cross-entropy:
$l(x,\ y) = -x\ln(y) - (1-x)\ln(1-y)$; in multi-classification problem, the popular choice is categorical cross-entropy: $l(x,\ y) = -\sum_{j=1}^{n} x^{[j]} \ln(y^{[j]})$, where n is the number of classes.

2.4 Gradient Descent

2.4.1 Basic Intuition of GD

Gradient descent (GD) is the foundation of a large scale of optimization algorithms in neural networks. Imagine that you stand on the top of a mountain and plan to go downhill, the most efficient choice is following the path that is the steepest. Namely, the most efficient method to find the minimum of J is to choose the gradient of J at the parameters $W,\ b$: $\nabla J(\hat{y},\ y)$ [1] Hence, we update the parameters at the learning rate α:

$$W^{[l]} = W^{[l]} - \alpha \frac{\partial J}{\partial W^{[l]}} \tag{2-4}$$

$$b^{[l]} = b^{[l]} - \alpha \frac{\partial J}{\partial b^{[l]}} \tag{2-5}$$

until $J(\hat{y},\ y)$ is small enough.

2.4.2 Basic Convergence Analysis of GD

It is not clear that whether GD can converge to the required global minimum rather than local minimum. However, a notable paper by Dauphin et al. [1] shows that based on empirical evidence, saddle points are bigger challenges instead of local minimum. Especially in high dimensional problems, saddle points are often surrounded by rugged plateau which dramatically slow down the speed of convergence. More detail about the landscape of neural network will be discussed in Subsection 5.2. Here we just give an basic analysis of GD algorithm.

Before the analysis, we need a criterion of convergence. There are mainly two criteria: the limit point is a stationary point and the convergence of the function value. Apparently, the latter one is more easy to achieve.

In Bertsekas et al. [2], proposition 1.2.1 to 1.2.4 give the convergence result of GD. Considering the large scale of use of fixed learning rate, proposition 1.2.3 regarding the Lipschitz value is the most helpful and well-known one. And its variant is shown in the following:

[1] $\nabla J(\hat{y},\ y) = (\frac{\partial J}{\partial W^{[1]}}, \cdots,\ \frac{\partial J}{\partial W^{[L]}},\ \frac{\partial J}{\partial b^{[1]}}, \cdots,\ \frac{\partial J}{\partial b^{[L]}})^T$

Proposition 2.1. *Let* $\{x_k\}$ *be a sequence generated by GD:* $x_{k+1} = x_k - \alpha d_k$*, where* $\{d_k\}$ *is the gradient of* $f(x_k)$*. For some* $L > 0,\ L \in R$ *that satisfies*[2]*:*

$$||\nabla f(x) - \nabla f(y)|| \leq L||x - y||, \quad \forall x,\ y \in \mathbb{R} \tag{2-6}$$

If $\forall k,\ d_k \neq 0$ *and*

$$\epsilon \leq \alpha \leq \frac{2 - \epsilon}{L} \tag{2-7}$$

where $\epsilon \in R^+$*. Then every limit point of* $\{x_k\}$ *is a stationary point of* f*.*

Proof. Using the descent lemma[3], we have:

$$f(x_k - \alpha d_k) - f(x_k) \leq \alpha \nabla f(x_k)^T d_k + \frac{1}{2}\alpha^2 L||d_k||^2 \tag{2-8}$$

$$= \alpha||d_k||^2(\frac{1}{2}\alpha L - 1) \tag{2-9}$$

The right-hand side of Equation 2-7 yields:

$$\frac{1}{2}\alpha L - 1 \leq -\frac{1}{2}\epsilon \tag{2-10}$$

Together with the left-hand of Equation 2-7:

$$f(x_k) - f(x_k - \alpha d_k) \geq \frac{1}{2}\epsilon^2||d_k||^2 \tag{2-11}$$

If $\{x_k\}$ converge to a non-stationary point x, then $||d_k||^2 < 0$ and $f(x_k) - f(x_k - \alpha d_k) \to 0$, so $||d_k|| \to 0$, which contradicts the assumption. Hence, every limit point of $\{x_k\}$ is stationary.

Q.E.D.

According to Proposition 2.1, we can choose the learning rate $\alpha = \frac{2}{L}$ and guarantee that GD will converge. Unfortunately, for optimization in neural network, such a global Lipschitz constant may not exist in most cases. until now, there seems to be no obvious way to fix the gap between the theorem and practice. In the article of Ruoyu Sun [3], a claim that may be sufficient for practitioners is proposed: if all parameters are bounded in each iteration, with a small and proper learning rate α, GD will converge.

[2] L is called the Lipschitz constant

[3] Details of descent lemma are in the Appendix 1

2.5 Forward Propagation

Forward propagation (FP) is a method to compute the predicted $\hat{y}$ from the input data. To illustrate the process of FP, we use the network shown in Figure 2.4. For the first layer, input of neurons is $A^{[0]} = (a^{[0](1)}, \cdots, a^{[0](m)}) \in \mathbb{R}^{n^{[0]} \times m}$, the output is $A^{[1]}$ and the cache is $Z^{[1]} = (z^{[1](1)}, \cdots, z^{[1](m)})$.

$$\begin{aligned} Z^{[1]} &= W^{[1]} A^{[0]} + b^{[1]} \\ A^{[1]} &= g^{[1]}(Z^{[1]}) \\ &\vdots \end{aligned} \tag{2-12}$$

Therefore, the general computation of the i^{th} layer is:

$$\begin{aligned} Z^{[l]} &= W^{[l]} A^{[l-1]} + b^{[l]} \\ A^{[l]} &= g^{[l]}(Z^{[l]}) \end{aligned} \tag{2-13}$$

2.6 Backward Propagation

Backward propagation (BP) is considered as an vital landmark in the development of neural networks. It highly boost the efficiency of the computation of gradient. Different from the FP, BP starts from the output layer of the network and goes through the network to the front. We denote that $dA^{[l]} \triangleq \frac{\partial J}{\partial A^{[l]}} \in \mathbb{R}^{n^{[l]} \times m}$, $dZ^{[l]} \triangleq \frac{\partial J}{\partial Z^{[l]}} \in \mathbb{R}^{n^{[l]} \times m}$, $dW^{[l]} \triangleq \frac{\partial J}{\partial W^{[l]}} \in \mathbb{R}^{n^{[l]} \times n^{[l-1]}}$ and $db^{[l]} \triangleq \frac{\partial J}{\partial b^{[l]}} \in \mathbb{R}^{n^{[l]} \times 1}$. For the L^{th} layer, we can

calculate the gradient by the following way[4] [5]:

$$\begin{aligned} dA^{[L]} &= \frac{\partial J}{\partial A^{[L]}} \\ dZ^{[L]} &= dA^{[L]} * g^{[L]'}(Z^{[L]}) \\ dW^{[L]} &= \frac{1}{m} dZ^{[L]} A^{[L-1]T} \\ db^{[L]} &= \frac{1}{m} dZ^{[L]} \\ dA^{[L-1]} &= W^{[L-1]T} dZ^{[L]} \\ dZ^{[L-1]} &= dA^{[L-1]} * g^{[L-1]'}(Z^{[L-1]}) \\ &\vdots \end{aligned} \tag{2-14}$$

Suppose the loss function of the output layer is binary cross-entropy:
$l(a^{[L]},\ y) = -y \ln(a^{[L]}) - (1-y) \ln(1-a^{[L]})$, then the BP process can be written as:

$$\begin{aligned} dZ^{[L]} &= A^{[L]} - y \\ dW^{[L]} &= \frac{1}{m} dZ^{[L]} A^{[L-1]T} \\ db^{[L]} &= \frac{1}{m} dZ^{[L]} \\ &\vdots \\ dA^{[l]} &= W^{[l]T} dZ^{[l+1]} \\ dZ^{[l]} &= (W^{[l]T} dZ^{[l+1]}) * g^{[l]'}(Z^{[l]}) \\ dW^{[l]} &= \frac{1}{m} dZ^{[l]} A^{[l-1]T} \\ db^{[l]} &= \frac{1}{m} dZ^{[l]} \end{aligned} \tag{2-15}$$

3 Problem in Neural Network

Though neural network is able to achieve numerous back breaking tasks, it is fairly challenging to well train a neural network. Among all the difficulties, gradient explosion

[4] the operator $*$ is element-wise multiplication

[5] the update of parameters in GD involve the complete data set, while in the variant of GD like SGD and mini-batch GD, only part of the data contribute to the update process in each iteration

and vanishing is the most famous one. In this section, we will discuss a few problems in training a neural network.

3.1 Explosion and Vanishing

Intuitively, using the computation like $Z^{[l]} = W^{[l]}A^{[l-1]}$ and $dA^{[L-1]} = W^{[L-1]T}dZ^{[L]}$ in FP and BP, if W is much larger than 1 or much smaller than 1, after several multiplication in the deep neural network the final result $Z^{[l]} = \prod_{l=1}^{L} W^{[l]}A^{[0]}$ will gradually tend to be infinity or zero. Here we further illustrate the explosion and vanishing with a simple problem from [4]:

Example 3.1. Suppose there is only one parameter $w.$ for a data set $\{x,\ y\}$ and a special architecture, the cost function is:

$$J(w) = (w^7 + 1)^2 \tag{3-1}$$

The plot of $J(w)$ is illustrated in Figure 3.1. The global minimum $w^* = -1$, a basin exists in the region of $[-1.05,\ -0.95]$. But in the region of $[-\infty,\ -1.05],\ [1.05,\ \infty]$, $J(w)$ is extremely steep, which is the explosion, and in the region of $[-0.95,\ 1.05]$, $J(w)$ is unusually flat, which is the vanishing. If the initial value w_0 falls into the basin, GD will converge quickly. If not, let's say $w_0 = 0.5$, then it would be terrible to traverse through the plateau with a tiny update in each iteration.

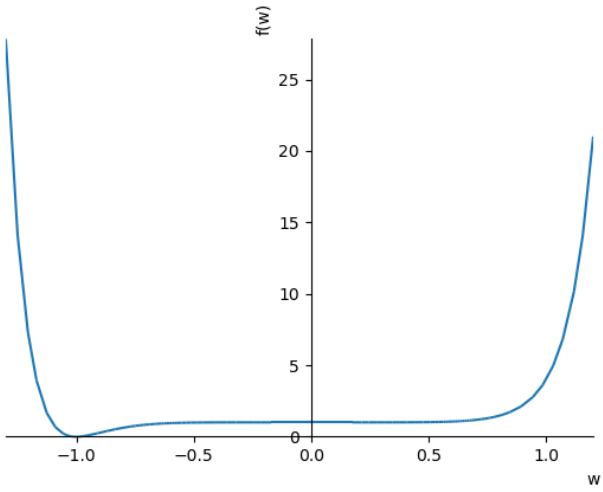


Figure 3.1: Plot of $J(w) = (w^7 + 1)^2$

3.2 Saddle Points in Non-convex Optimization

Though there seems to be no rigorous proof of the prevalence of saddle points in high dimensional non-convex optimization, one line of the evidence is carried out astonishingly from statistical physics. It suggests that among a myriad of critical points, most are likely to be saddle points. To understand the landscape in the neural network optimization, we first introduce the Hessian Matrix to pave the way for further discussion.

3.2.1 Hessian Matrix

Definition 3.1. Suppose $f : \mathbb{R}^n \to R$, all second partial derivatives of f exist and are continuous over the domain. The Hessian Matrix $\mathbf{H} \in \mathbb{R}^{n \times n}$ of f is defined and arranged as follow:

$$\mathbf{H} = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{pmatrix} \tag{3-2}$$

If the second partial derivatives are continuous, then the order of partial derivatives does not matter. Thus, **H** is a symmetric matrix.

Since the Hessian Matrix is symmetric, the eigenvalues are real numbers, i.e. $\lambda_i \in R$. For a cost function $J(\theta)$, where $\theta \in \mathbb{R}^n$ represents the high dimensional variable. The properties of critical points[6] can be described by the eigenvalues of the Hessian.

1. If $\forall \lambda_i \in R^+$, i.e. Hessian is positive definite, then the critical point is a local minimum.

2. If $\forall \lambda_i \in R^-$, i.e. Hessian is negative definite, then the critical point is a local maximum.

3. If $\forall \lambda_i \neq 0$, some are positive and others are negative, then the critical point is a (horse) saddle point with min-max structure.

4. If Hessian matrix is singular, i.e. $|\mathbf{H}| = 0$, then the critical point is called degenerate critical point and it is a monkey saddle point.

3.2.2 The Prevalence of Saddle Points

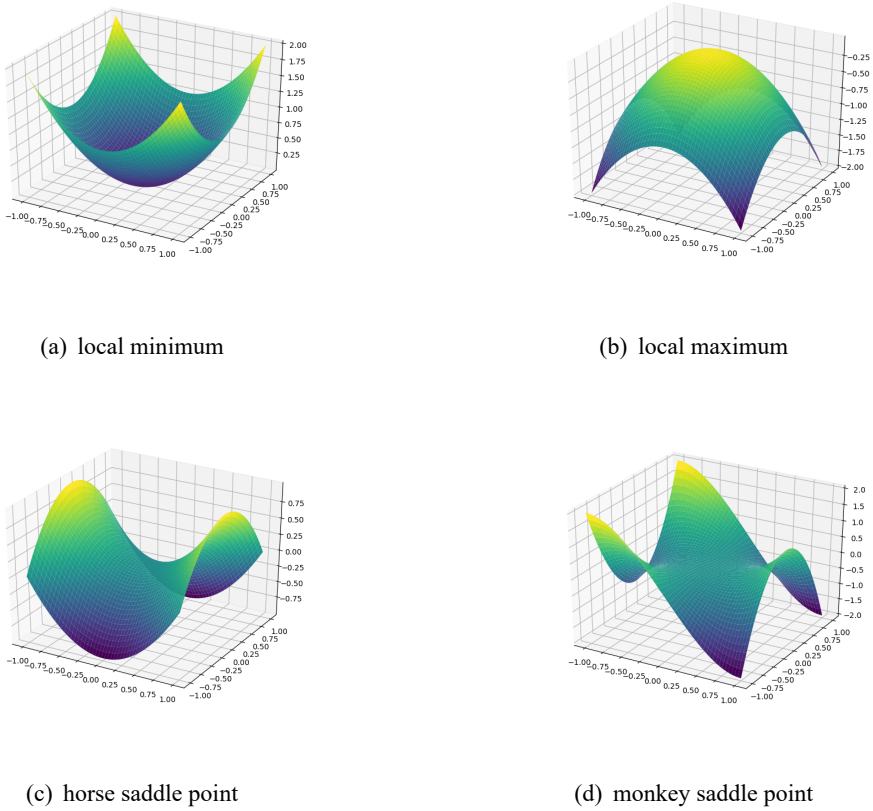
In the early version of neural network algorithms, parameters are initialized with standard Gaussian noise. The article by Bray et al. [5] computes the average number of critical points of a Gaussian distribution on a high dimensional space. They derive the distribution of critical points with a relation between α and ϵ, where α is the fraction of negative eigenvalues of the Hessian at the critical points, ϵ is the error at the critical points. In a Gaussian field ϕ defined over volume V of an N-dimensional Euclidean space. The normalized density of eigenvalues defined as:

$$\rho(\lambda) = \frac{1}{N} \sum_{i=1}^{N} \delta(\lambda - \lambda_i) \tag{3-3}$$

The average of eigenvalues is defined as:

$$\bar{(}\lambda) = \int \lambda \rho(\lambda) d\lambda \tag{3-4}$$

[6]Critical points are points θ where $\nabla J(\theta) = 0$.



(a) local minimum

(b) local maximum

(c) horse saddle point

(d) monkey saddle point

Figure 3.2: Different types of critical point

And it can be computed via:

$$\bar{\lambda}(\epsilon) = 2\frac{f'(0)\epsilon}{f''(0)P} \tag{3-5}$$

where P is defined as follow:

$$P = \frac{f'(0)^2}{f''(0)^2} + \frac{f(0)}{f''(0)}\left(1 - \frac{2}{N}\right) \approx \frac{f'(0)^2}{f''(0)^2} + \frac{f(0)}{f''(0)} \tag{3-6}$$

Finally, the relation between α and ϵ is given by:

$$\frac{2}{\pi}\int_{\frac{\bar{\lambda}}{2\sqrt{f''(0)}}}^{1} (1-u^2)^{\frac{1}{2}} du = \alpha \tag{3-7}$$

where u just a temporary variable in the integration computation.

In the $\epsilon - \alpha$ graph, there is a global minimum at $\alpha = 0,\ \epsilon = \epsilon_{min}$ and a global maximum at $\alpha = 1,\ \epsilon = \epsilon_{max}$. Other critical points a located on a monotonically increasing curve as α ranges from 0 to 1. Some experiments validating such proposal

is finished by Dauphin et al. [1] in Figure 3.3. This implies that local minimums, i.e. critical points that $\alpha \to 0$, are more closer to global minimum, while the majority of critical points that have high error are likely to have a large percentage of negative eigenvalues, i.e. most of critical points are saddle points.

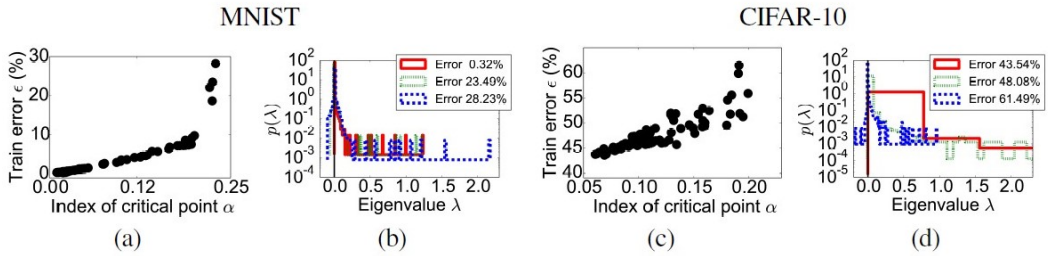


Figure 3.3: (a) and (c) show how critical points are distributed in the $\epsilon - \alpha$ plane. (b) and (d) plot the distributions of eigenvalues of the Hessian at three different critical points.

3.3 Learning Rate Selection

The choice of stepsize has long been a puzzle to practitioners in neural networks. In fact, it is not peculiar to GD but a common problem in optimization. If the learning rate is too large, with high probability GD will diverge. If learning rate is too small, the time GD takes to converge is unbearable. In practice, the most widely used scheme is setting learning rate as a small constant like 0.1 or 0.01. The drawback of such scheme is obvious: it does not guarantee the convergence of GD.

First we review the general optimization problem:

$$x_{k+1} = x_k - \alpha_k d_k$$

We list other commonly used scheme in the following.

3.3.1 Goldstein Rule

Goldstein rule is the first effective rule for stepsize selection in general optimization problem that does not depend on linear minimization. Let $\sigma \in (0,\ 0.5),\ \alpha_k$ satisfied:

$$\sigma \leq \frac{f\left(x^k + \alpha^k d^k\right) - f\left(x^k\right)}{\alpha^k ||d^k||^2} \leq 1 - \sigma \tag{3-8}$$

3.3.2 Armijo Rule

It is natural to devise a scheme that successively reduce α, when $f(x_{k+1}) < f(x_k)$ is not satisfied. The Armijo rule is exactly based on the this devisal. Here, let $s \in R,\ \beta,\ \sigma \in [0,\ 1],\ \alpha_k = \beta^{m_k} s$, where m_k is the first nonnegative integer satisfies:

$$f(x_k) - f(x_k - \beta^{m_k} s d_k) \geq -\sigma \beta^{m_k} s \nabla ||d_k||^2 \tag{3-9}$$

Usually $\sigma \in [10^{-5},\ 10^{-1}],\ \beta \in [0.1,\ 0.5],\ s = 1$.

3.3.3 Diminishing Learning Rate

During the process of optimization, we let $\alpha_k \to 0$. The primary downside of such scheme is that after multiple iterations the stepsize may be too small to descent. So an additional requirement is added:

$$\sum_{k=0}^{\infty} \alpha_k = \infty \tag{3-10}$$

4 Tricks in Neural Network

Training neural network to achieve a decent performance in ordinary method is challenging. In recent years, a growing number of ultra deep neural networks, like VGG-16 with 16 layers and over 100 million parameters, can be easily trained. The huge leap attributes to the diligent theorists who devised various sophisticated tricks boosting the training efficiency and addressing some of the problems mentioned in Section 3.

4.1 A Good Initialization is All You Need [7]

From the experience in Subsubsection 3.2.2, a initialization with common Gaussian distribution appears to an obstacle to convergence. Romero et al. [7] states that for deep neural net with too many layers and especially those with uniform normalization, it

[7] The title of this section is quoted from Mishkin et al. [6] whose title had a profound influence on our study of the neural network.

is hard to train using BP. Thus, we hereby conclude a number of effective initialization methods.

4.1.1 Xavier Initialization

In [8], Glorot et al. proposed a adoption of standard Gaussian initialization which was called "Xavier" initialization by Jia et al. [9]. Xavier initialization aims to improve the performance of neural network using sigmoid activation and log-likelihood loss function:

$$\begin{aligned} E(W^{[l]}) &= 0, \\ Var(W^{[l]}) &= \frac{2}{n^{[l-1]} - n^{[l]}} \end{aligned} \tag{4-1}$$

Proof. For a activation function g with $f'(0) = 1$, like sigmoid $\sigma(z)$, if we use the notation in Equation 2-13 and Equation 2-15:

$$\begin{aligned} Z^{[l]} &= W^{[l]}A^{[l-1]} + b^{[l]} \\ dZ^{[l]} &= (W^{[l]T}dZ^{[l+1]}) * g^{[l]'}(Z^{[l]}) \\ dW^{[l]} &= dZ^{[l]}A^{[l-1]T} \end{aligned}$$

We assume that the weights are independent and the distribution of input X are the same. Then $g^{[l]'}(Z_k^l) \approx 1$, Using the chain rule we have:

$$Var(A^{[l]}) = Var(X)\prod_{l'=0}^{l-1} n^{[l'-1]}Var(W^{[l']}) \tag{4-2}$$

$$Var(dZ^{[l]}) = Var(dZ^{[L]})\prod_{l'=L}^{l} n^{[l']}Var(W^{[l']}) \tag{4-3}$$

To keep information flowing in FP and BP, we would like to have:

$$\forall(l,\ l'),\ Var(A^{[l]}) = Var(A^{[l']}) \tag{4-4}$$

$$\forall(l,\ l'),\ Var(dZ^{[l]}) = Var(dZ^{[l']}) \tag{4-5}$$

So:

$$\forall l,\ n^{[l'-1]}Var(W^{[l']}) = 1 \tag{4-6}$$

$$\forall l,\ n^{[l']}Var(W^{[l']}) = 1 \tag{4-7}$$

To combine the two equations above, we may have:

$$Var(W^{[l']}) = \frac{2}{n^{[l-1]} - n^{[l]}} \tag{4-8}$$

Q.E.D.

For Gaussian initialization, Xavier suggest to have:

$$W \sim N(0,\ \frac{2}{n^{[l-1]} - n^{[l]}}) \tag{4-9}$$

Though rarely in use, a normalized version of Xavier initialization is provided in [8]:

$$W \sim U(-\frac{\sqrt{6}}{\sqrt{n^{[l-1]} + n^{[l]}}},\ \frac{\sqrt{6}}{\sqrt{n^{[l-1]} + n^{[l]}}}) \tag{4-10}$$

4.1.2 Kaiming Initialization

He et al. [10] extended the Equation 4-1 to ReLU activation and achieved a better performance. In this article, a modification of ReLU, the Parametric Rectified Linear Unit (PReLU)[8] was proposed as well. Due to the lack of $g'(0) = 1$ in ReLU, "Xavier" failed to pursue a convergence in deep neural networks. Kaiming initialization is set as follow:

$$\begin{aligned} E(W^{[l]}) &= 0, \\ Var(W^{[l]}) &= \frac{2}{n^{[l]}} \end{aligned} \tag{4-11}$$

Proof. We use the notation in Equation 2-13 and Equation 2-15:

$$\begin{aligned} Z^{[l]} &= W^{[l]}A^{[l-1]} + b^{[l]} \\ dA^{[l]} &= W^{[l]T}dZ^{[l+1]} \end{aligned}$$

And we review the ReLU: $g(z) = max(0,\ z)$. Let the initialization in parameters and input are mutually independent and share the same distribution. Then we have:

$$\begin{aligned} Var(Z^{[l]}) &= n^{[l]}Var(W^{[l]}A^{[l-1]}) \\ &= n^{[l]}(Var(W^{[l]})Var(A^{[l-1]}) + Var(W^{[l]})E(A^{[l-1]})) \end{aligned} \tag{4-12}$$

[8]The detail of PReLU will be discuss in Appendix 2

Note that $g(z) = max(0,\ z)$, this leads to $E(g(z)) = \frac{1}{2}E(z),\ Var(g(z)) = \frac{1}{4}Var(z)$. Therefore, Equation 4-12 can be converted to:

$$Var(Z^{[l]}) = \frac{1}{2}n^{[l]}Var(W^{[l]})Var(Z^{[l-1]}) \tag{4-13}$$

Putting all layers together, we have:

$$Var(Z^{[L]}) = Var(Z^1)\prod_{l=2}^{L}\frac{1}{2}n^{[l]}Var(W^{[l]}) \tag{4-14}$$

Similar to the proof in Xavier initialization, we hope the maintain the variance in the FP to avoid the possible explosion or vanishing. So,

$$\forall l,\ \frac{1}{2}n^{[l]}Var(W^{[l]}) = 1 \tag{4-15}$$

Likewise in the process of BP, due to the property of ReLU function,

$$Var(dA^{[1]}) = Var(dA^{L})\prod_{l=1}^{L}\frac{1}{2}n^{[l]}Var(W^{[l]}) \tag{4-16}$$

and the corresponding condition is the same as Equation 4-15. **Q.E.D.**

4.1.3 Orthogonal Initialization

Based on deep linear neural networks, Saxe et al. [11] exhibits a class of random orthogonal initialization which enjoys depth independent learning time in linear networks. Orthogonal initialization is implemented by choosing all weights to be orthogonal matrix:

$$(W^{[l]})^T W^{[l]} = I \tag{4-17}$$

The implementation can be divided into two steps:

1. Fill the weights with standard Gaussian distribution.

2. Decompose the weights matrix using singular value decomposition (SVD) and replace weights with one of the components.

Saxe et al. suggest that an approriate condition on weights for generating fast learning speed would be dynamical isometry[9]. Saxe et al. [11] illustrate the answer for depth independence by visualize the eigenvalue spectrum of a random orthogonal matrix in Figure 4.1, which is exactly a unit circle. Moreover, its singular values are all exactly 1.

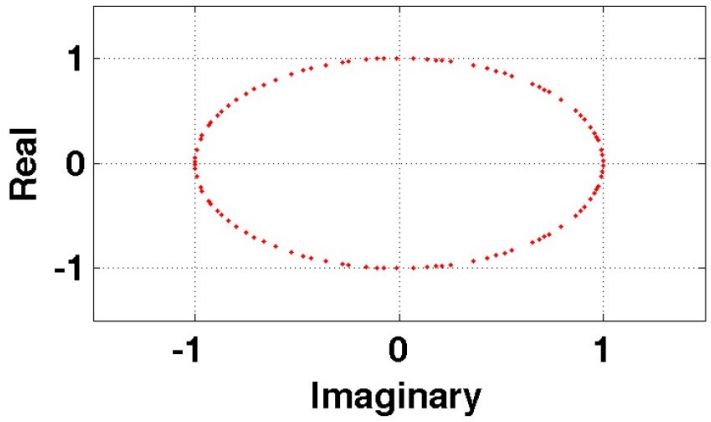


Figure 4.1: The eigenvalue spectrum of a random orthogonal matrix $W \in \mathbb{R}^{100\times100}$

Then in the linear neural networks, the input-output Jacobian is:

$$J = \prod_{l=1}^{L} W^{[l]} \tag{4-18}$$

Under orthogonal initialization in each layer, J is a orthogonal matrix and thus the initialization achieve dynamical isometry. Still the reason for dynamical isometry explaining the depth independence will be studied in the future work.

4.1.4 Layer-Sequential Unit-Variance Initialization (LSUV)

As far as we aware, part from the Kaiming initialization, mentioned in Subsubsection 4.1.2, there is no extension of Xavier initialization to other nonlinear function other than ReLU. Mishkin et al. [6] provide a general and simple initialization method that works well with different activation functions:

[9]Dynamical isometry means that the input-output Jacobian $J = \frac{\nabla A^{[L]}}{\nabla A^{[0]}}$ has all singular values close to 1.

1. Initialize the weights using orthogonal initialization.

2. Scale the variance of output of each layer for each mini-batch [10] to be 1.

Though the explanation of why LSUV outperform other initialization methods is not given in the article, a series of comparison empirically show its advantage in neural networks initialization problem.

4.1.5 Comparison between initialization

We implemented a simple CNN model based on keras to examine the performance of random, xavier and he initialization. The model trained the MNIST data set [12] to classify the handwriting of Arabic numbers. The CNN model is constructed with two convolution layers, two max pooling layers and two fully connected layers. Activations use ReLU and the output layer uses Softmax. Due to the limitation of the version of keras and time limit, LSUV is not implemented. The code is attached in the "/code/initialize.py". The size of data set is $M = 60000$, we set $m = 64$. The results are demonstrated in Figure 4.2 and Figure 4.3.

[10] Mini-batch means a small proportion of the data set used in one iteration, which will be introduced in Subsubsection 4.2.3.

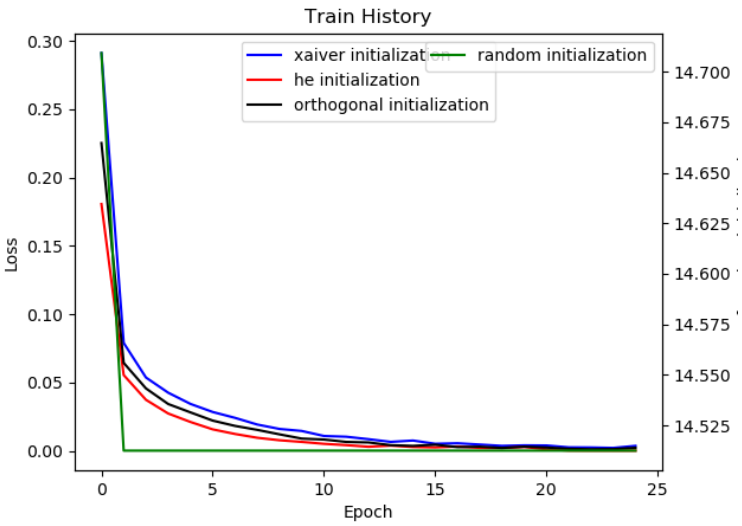


Figure 4.2: The loss in training of different initialization methods, only random initialization corresponds to the right y axis

```
Using random initialization:
loss: 14.4869439453125
accuracy: 0.10119999945163727
**********
Using xavier initialization:
loss: 0.03422860299189306
accuracy: 0.9907000064849854
**********
Using he initialization:
loss: 0.03117609064016858
accuracy: 0.992900013923645
**********
Using orthogonal initialization:
loss: 0.02801127192603617
accuracy: 0.9926999807357788
```

Figure 4.3: The loss and accuracy on test set of different initialization methods

Surprisingly, CNN using random initialization did not converge and did not learn after the second epoch. Others have a great convergence, he initialization is a bit faster than the rests since it is tailored for ReLU.

4.2 Gradient Descend Variants

There are two variants of GD improving the training efficiency. In brevity, batch describes the data set used in an iteration[11], epoch describes a traversal through the whole data set. GD introduced in Subsection 2.4 is the basic form of GD. If we unroll the vectorization in Equation 2-4, we can have:

$$\begin{aligned} dW^{[l]} &= \frac{1}{m}\sum_{i=1}^{m} dZ^{[l]}_{\bullet,\,i} A^{[l-1]T}_{\bullet,\,i} \\ db^{[l]} &= \frac{1}{m}\sum_{i=1}^{m} dZ^{[l]}_{\bullet,\,i} \end{aligned} \tag{4-19}$$

The variants take on different convergence speed and accuracy by choosing different m.

4.2.1 Batch Gradient Descent (BGD)

Batch gradient descent, computes the gradient for the whole data set, i.e. m is the size of the data set M. Despite the high accuracy resulted from taking full advantage of data set, BGD failed to manage the online algorithms due to the considerable training time.

4.2.2 Stochastic Gradient Descent (SGD)

In comparison to BGD, SGD is competent for computation new data on the fly. In each iteration, SGD updates the parameters using only one piece of data $(x^{(i)},\ y^{(i)})$, i.e. $m = 1$:

$$\begin{aligned} dW^{[l]} &= \sum_{i=1}^{m} dZ^{[l]} A^{[l-1]T} \\ db^{[l]} &= \sum_{i=1}^{m} dZ^{[l]} \\ dZ^{[l]},\, A^{[l]} &\in \mathbb{R}^{n^{[l]}\times 1} \end{aligned} \tag{4-20}$$

[11] In an iteration, all parameters are updated.

Since only one sample is applied to the computation of gradient, the speed of SGD is fast enough for daily application. On the other hand, optimizing for one sample does not guarantee the convergence to a minimum and a descent of cost in each iteration. It is therefore SGD suffers a severe fluctuation and may wander around the minimum.

4.2.3 Mini-batch Gradient Descent (MGD)

To balance the merit of both BGD and SGD, mini-batch gradient descent choose an appropriate $m = 2^k,\ k \in R^+$. Usually, m will be chosen as 64, 128 and 256. Taking the best of both worlds, MGD descent has a fast and stable convergence.

4.2.4 Comparison Between Variants

To illustrate the different between batch BGD, SGD and MGD, we implement a multi-classification neural network based on the tensorflow framework. The architecture of deep neural network with two hidden layers using ReLU and the output layer using Softmax, Adam optimization mentioned in Subsubsection 4.3.6 is applied to pursue better convergence. Due to the limitation of the old version of tensorflow, only Xavier initialization is available. The code attached in the ” /code/test.py ”. The size of data set is $M = 1080$, we set $m = 1080,\ 1,\ 64$ respectively for three variants. The results are shown in the following , see Figure 4.4,Figure 4.5,Figure 4.6,Figure 4.7,Figure 4.8,Figure 4.9.

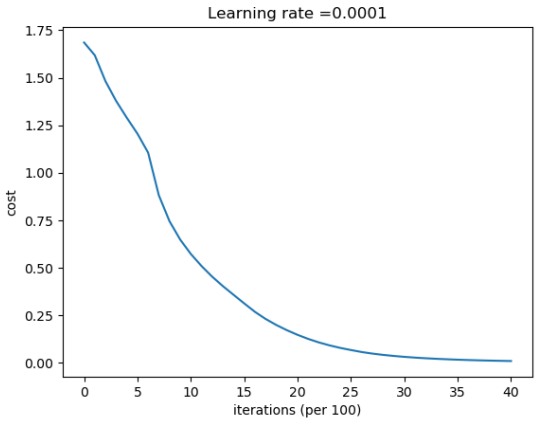


Figure 4.4: The loss of BGD during training

```
Cost after epoch 0: 1.811549
Cost after epoch 1000: 0.648244
Cost after epoch 2000: 0.171951
Cost after epoch 3000: 0.036816
Cost after epoch 4000: 0.011134
CPU executing time = 1275.3314905 s
Parameters have been trained!
Train Accuracy: 1.0
Test Accuracy: 0.8833333
```

Figure 4.5: The result of training

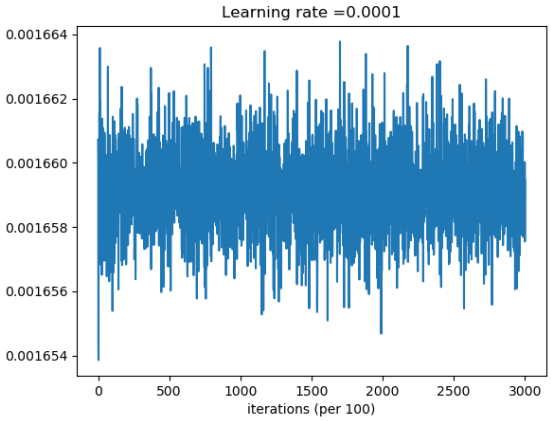


Figure 4.6: The loss of SGD during training

```
Cost after epoch 250: 1.791872
Cost after epoch 260: 1.791866
Cost after epoch 270: 1.791864
Failed to converge!
CPU executing time = 709.3755229 s
Parameters have been trained!
Train Accuracy: 0.16666667
Test Accuracy: 0.16666667
```

Figure 4.7: The result of training

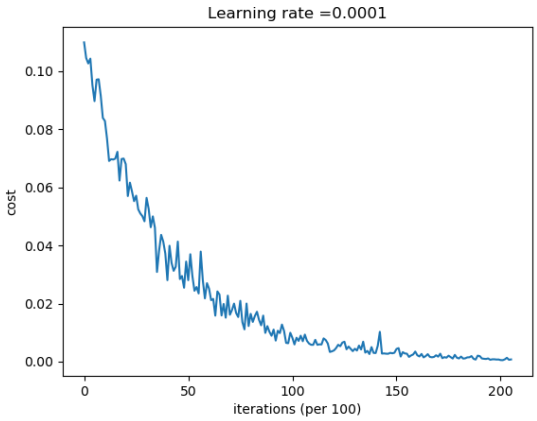


Figure 4.8: The loss of MGD during training

```
Cost after epoch 1000: 0.029091
Cost after epoch 1100: 0.018433
Cost after epoch 1200: 0.011889
CPU executing time = 341.1836363 s
Parameters have been trained!
Train Accuracy: 1.0
Test Accuracy: 0.89166665
```

Figure 4.9: The result of training

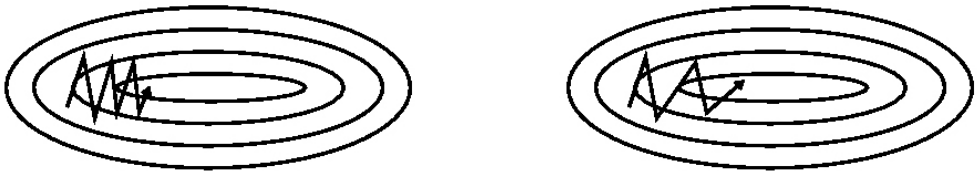
Unfortunately, under the testing condition, SGD did not converge but stuck in some kind of local minimum or saddle point. Comparing MGD and BGD, to achieve the same accuracy of 99%, MGD is much more faster but some glitches can be observed in the cost, which means MGD dose not guarantee descent in each iteration like SGD. In all, the simple comparison roughly demonstrate the advantage of MGD in the optimization process.

4.3 Optimization Algorithms

Mini-batch GD does improve the convergence speed in deep neural networks, but the aforementioned challenges like saddle points, learning rate selection are still need to be addressed. We summarize some prevailing optimization algorithms as follow.

4.3.1 Momentum

Momentum method is one of the earliest simple and robust modification to pursue faster learning by Qian et al. [13]. When the cost function is similar to a long and narrow valley like Figure 4.10, most gradient is perpendicular to the long axis, the the back and forth track would drastically slows down the learning speed.



(a) Without momentum (b) With momentum

Figure 4.10: A long and narrow valley situation. From [14]

In this situation, we are eager to keep the track parallel to the long axis by some kind of force. In physics, it is called the momentum, and thus the terminology comes into being. Momentum updates the parameters in the following way:

$$\begin{aligned} v_t &= \beta v_{t-1} - \alpha dW_t^{[l]},\ v_0 = 0 \\ W_t^{[l]} &= W_{t-1}^{[l]} + v_t \end{aligned} \tag{4-21}$$

The recursive definition of v_t is called the exponentially decaying average.

The relation between momentum method and its physics background is readily comprehensible. When a ball moving in a field with a fraction w.r.t. the speed, then the Newtonian equation is:

$$m\frac{d^2x}{dt^2} + \mu\frac{dx}{dt} = -\nabla E(x) \tag{4-22}$$

where m is the mass, μ is the fraction coefficient and $E(x)$ is the potential energy. Since $\frac{dx}{dt} = \frac{x_{t+1}-x_t}{\Delta t}$, Equation 4-22 can be rewritten as:

$$\begin{aligned} m\frac{(x_{t+1}-x_t)-(x_t-x_{t-1})}{(\Delta t)^2} + \mu\frac{x_{t+1}-x_t}{\Delta t} &= -\nabla E(x) \\ x_{t+1}-x_t = -\frac{(\Delta t)^2}{m+\mu\Delta t}\nabla E(x) + \frac{m}{m+\mu\Delta t}(x_t - x_{t-1}) \end{aligned} \tag{4-23}$$

If we set $\frac{(\Delta t)^2}{m+\mu\Delta t} = \alpha,\ \frac{m}{m+\mu\Delta t} = \beta$, then we obtain the Equation 4-21.

Here, we briefly demonstrate the convergence analysis of momentum method. Using the Hessian $\nabla E(x_t) = Hx_t$ and similarity transformation $H = Q^T DQ,\ Q^T Q = I$, we can convert Equation 4-23 into:

$$\begin{aligned} x_{t+1} &= ((1+\beta)I - \alpha Q^T DQ)x_t - \beta x_{t-1} \\ Q^T x_{t+1} &= (1+\beta)IQ^T x_t - \alpha Q^T Q^T DQx_t - \beta Q^T x_{t-1} \end{aligned} \tag{4-24}$$

We denote $x_t = x'_t$, then the element-wise equation is:

$$x'_{i,\ t+1} = ((1+\beta) - \alpha d_i)x'_{i,\ t} - \beta x'_{i,\ t-1} \tag{4-25}$$

Then a matrix representation of this equation is:

$$\begin{aligned} \begin{pmatrix} x'_{i,\ t} \\ x'_{i,\ t+1} \end{pmatrix} &= A_i \begin{pmatrix} x'_{i,\ t-1} \\ x'_{i,\ t} \end{pmatrix} \\ A_i &= \begin{pmatrix} 0 & 1 \\ -\beta & 1+\beta-\alpha d_i \end{pmatrix} \end{aligned} \tag{4-26}$$

The convergence of momentum is determined by the linear system in Equation 4-26. Recall the knowledge in numerical analysis, a theorem w.r.t. the convergent matrix is stated in Theorem 7.17 in Burden et al. [15]:

Theorem 4.1. *A is a convergent matrix*[12] $\iff$ *the spectrum radius $\rho(A)$*[13] *satisfies*

$$\rho(A) < 1 \tag{4-28}$$

.

A proposition given in Appendix B in Qian et al. [13] provide the theoretical foundation of the convergence.

[12] We call a matrix A convergent if

$$\lim_{k\to\infty} \left(A^k\right)_{ij} = 0, \quad \text{for each } i = 1, 2, \ldots, n \text{and } j = 1, 2, \ldots, n \tag{4-27}$$

.

[13] Spectrum radius is defined as $\rho(A) = \max(|\lambda|)$

Proposition 4.1. *Let $\lambda_{i,\,1},\ \lambda_{i,\,2}$ are the eigenvalues of A_i. The system in Equation 4-26 converges if and only if*

$$-1 < \beta < 1,\ 0 < \alpha d_i < 2 + 2\beta \tag{4-29}$$

4.3.2 Nesterov Accelerated Gradient (NAG)

NAG in Nesterov [16] is a optimization method solving convex problems with convergence rate $O(1/K^2)$, where K is the number of iterations. NAG is virtually similar to momentum method in spite of two differences. First, learning rate is derived from the Armijo rule with $\beta = 2,\ \sigma = 0.5$ in Subsubsection 3.3.2. The momentum constant β is formulated in a recursive way that $\beta_t \sim \frac{t+4}{2}$, where t is the index of iteration. Second, a subtle update rule is applied in NAG. Considering the fact that the former difference just an additional condition to guarantee the convexity, such formulation seems trivial in the common non-convex optimization. Instead, we manually choose the parameters β and α in advanced and focus more on the later aspect:

$$\begin{aligned} v_t &= \beta v_{t-1} - \alpha \nabla J(W_{t-1} + \beta W_{t-1}),\ v_0 = 0 \\ W_t &= W_{t-1} + v_t \end{aligned} \tag{4-30}$$

To the best of our knowledge, there have been no attempts to generalize the convergence analysis of NAG to non-convex problem. But a empirical experiment in Sutskever's PhD thesis [17].

Intuitively, to comprehend the improvement of NAG, imagine SGD as a ball rolling downhill (like in the physics background of momentum). In a rough terrain, blindly rolling with gradient seems to be silly, because there is no forecast when the ball suddenly face a upward slope. As shown in Figure 4.11, NAG is more stable than momentum due to the backward correction.

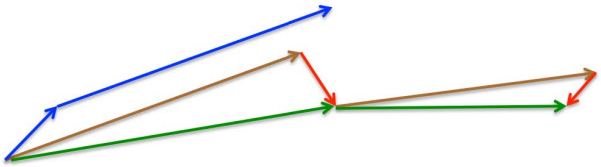


Figure 4.11: NAG illustration: blue vector represents the momentum method, green vector represents the NAG with the brown component is the gradient and the red component is the correction.

4.3.3 RMSprop

It is interesting that RMSprop is not published in paper but propose in a Coursera course by Hinton et al. [18]. RMSprop adopts the idea of only using the sign of the gradient by dividing the gradient with the size of it:

$$\begin{aligned} u_t &= \gamma u_{t-1} + (1-\gamma)(dW_t^{[l]})^2,\ u_0 = 0 \\ W_{t+1}^{[l]} &= W_t^{[l]} - \frac{\eta}{\sqrt{u_t + \epsilon}} dW_t^{[l]} \end{aligned} \tag{4-31}$$

Exponentially decaying average u_t is applied to the second moment of gradient. In brevity, we denote such average as $RMS(dW_t^{[l]})$. ϵ is a smoothing term that prevents division by zero and usually set to 10^{-8}. In the lecture, Hinton suggests $\gamma = 0.9$.

4.3.4 Adagrad

Duchi et al. [19] make a vivid metaphor that some infrequent but predictive features are needles in haystacks. Adagrad is a subgradient method that crafts domain-specific weightings by performing large updates for infrequent features and small updates for frequent ones.

$$W_t^{[l]} = W_{t-1}^{[l]} - \frac{\eta}{\sqrt{\sum_{t'=1}^{t} (dW_{t'}^{[l]})^2 + \epsilon}} dW_t^{[l]} \tag{4-32}$$

Note that the update is divided by the sum of the squared gradients, the learning rate will shrink to vanishing with the iteration. To address this flaw, an adaption called Adadelta is illustrated in the following.

4.3.5 Adadelta

Adadelta presented in Zeiler et al. [20] aims to improve Adagrad in two aspect: the diminishing learning rate and the manual tunning learning rate.

First, instead of simply summing up the squared gradients, a accumulation over window method is introduced, which turns out to be exactly the RMSprop in Subsubsection 4.3.3. Second, the article suggests that as a update of W, ΔW should have the same unit as W. However, in usual optimization method, this is not the case:

$$\text{units of}\Delta W \propto \text{units of}dW \propto \frac{\partial J}{\partial W} \propto \frac{1}{\text{units of}W} \tag{4-33}$$

Comparing with the Newton's method in Chapter 10.2 in Burden et al. [15], the Hessian is used to maintain the same unit:

$$\Delta W \propto H^{-1}dW \propto \frac{\frac{\partial J}{\partial W}}{\frac{\partial^2 J}{\partial W^2}} \propto \text{ units of } W \tag{4-34}$$

The term $\frac{1}{\frac{\partial^2 J}{\partial W^2}} = \frac{\Delta W}{\frac{\nabla J}{\nabla W}}$ can be added to obtain the match of unit. The unknown ΔW in the current iteration will be substituted by the previous one. Since the denominator use the RMSprop, the numerator should take the identical form:

$$\begin{aligned} v_t &= -\frac{RMS(v_{t-1})}{RMS(dW_t^{[l]})} \\ W_{t+1}^{[l]} &= W_t^{[l]} + v_t \end{aligned} \tag{4-35}$$

4.3.6 Adam

The name of Adam comes from adaptive moment estimation in Kingma et al. [21] It is a memory saving and easy to compute optimization method combining both the advantages of RMSprop and momentum. The pseudo code of Adam is shown in Algorithm 1.

Using the definition in mathematical statistic, the Exponential decaying average calculates the mean and uncentered variance of gradients, i.e. the 1st moment and the 2nd raw moment. Due to the initialization of zero which is not there actual value, the

Algorithm 1 Adam. Note that all operators are element-wise. The recommended hyperparameters are learning rate $\alpha = 0.001$, momentum constant $\beta = 0.9$, RMSprop constant $\gamma = 0.999$, smoothing term $\epsilon = 10^{-8}$.

Require: α: Stepsize

Require: $\beta,\ \gamma \in [0,\ 1)]$: Exponential decay rates for moment estimation

Require: $J(W)$: Cost function with parameters W

Require: W_0: Initial parameter

$m_0 \leftarrow 0$ (Initialize 1st moment vector)

$v_0 \leftarrow 0$ (Initialize 2nd moment vector)

$t \leftarrow 0$ (Initialize time step)

while W_t not converged **do**

$t \leftarrow t + 1$

$g_t \leftarrow \nabla J(W_{t-1})$(Get gradients w.r.t. cost function)

$m_t \leftarrow \beta m_{t-1} + (1 - \beta) g_t$ (Update biased first moment estimate)

$v_t \leftarrow \gamma v_{t-1} + (1 - \gamma) g_t^2$ (Update biased second moment estimate)

$\hat{m}_t \leftarrow \frac{m_{t-1}}{1-\beta^t}$ (Compute bias-corrected first moment estimate)

$\hat{v}_t \leftarrow \frac{v_{t-1}}{1-\gamma^t}$ (Compute bias-corrected second raw moment estimate)

$W_t \leftarrow W_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$ (Update parameters)

end while

return W_t (Resulting parameters)

initialization bias correction. Calculate the recursive definition of m_t, we can obtain:

$$m_t = \beta^t m_0 + (1-\beta)\sum_{i=1}^{t}\beta^{t-i}g_i \tag{4-36}$$

Initialize $m_0 = 0$, and we have:

$$m_t = (1-\beta)\sum_{i=1}^{t}\beta^{t-i}g_i \tag{4-37}$$

We want the expectation of m_t and g_t is equivalent.

$$E(m_t) = (1-\beta)\sum_{i=1}^{t}\beta^{t-i}E(g_t) + \zeta \tag{4-38}$$

$$= (1-\beta^t)E(g_t) + \zeta \tag{4-39}$$

If $E(g_i)$ does not change with time, then $\zeta = 0$. Even if $E(g_i)$ varies, an appropriate β can keep ζ involved the past gradients small. Therefore, adding a $1-\beta^t$ in the denominator can correct the bias.

A analysis of convergence only for convex cost function is given in the article [21]. Under the definition of regret:

$$R(T) = \sum_{t=1}^{T}(J(W_t) - J(W^*)) \tag{4-40}$$

where $W^* = \arg\min_{W\in(W_1,\cdots,W_T)}\sum_{t=1}^{T}J(W)$. The theorem and proof is too obscure, hence we simply illustrate a corollary showing the bound of regret.

Corollary 4.1. *Assume that the function J has bounded gradients, $||\nabla J(W)||_2 \leq G$, $||\nabla J(W)||_\infty \leq G_\infty$, $\forall W$, and distance between any W_t is bounded, $||W_m - W_n||_2 \leq D$, $||W_m - W_n||_\infty \leq D_\infty$, $\forall m,\ n \in (1,\cdots,T)$. Adam achieve the following guarantee:*

$$\forall T \geq 1,\ \frac{R(T)}{T} = O(\frac{1}{\sqrt{T}}) \tag{4-41}$$

Therefore, the regret bound $O(\sqrt{T})$ is obtained.

4.3.7 AdaMax

AdaMax is a simple and stable extension of Adam is attached in the same article [21]. While L_2 Norm is numerically complex, AdaMax uses L_p Norm for the 2nd raw moment, and let $p \to \infty$:

$$\begin{aligned} v_t &= \gamma^{\infty} v_{t-1} + (1-\gamma^{\infty}) g_t^{\infty} \\ &= max(\gamma v_{t-1},\ |g_t|) \end{aligned} \tag{4-42}$$

The derivation in Equation 4-42 is similar to L_∞ Norm. In such adaption, even though the initialization of $v_0 = 0$, there is no need to correct the bias. See Algorithm 2 for a complete pseudo code.

Algorithm 2 AdaMax. Note that all operators are element-wise. The recommended hyperparameters are learning rate $\alpha = 0.002$, momentum constant $\beta = 0.9$, RMSprop constant $\gamma = 0.999$, smoothing term $\epsilon = 10^{-8}$.

Require: α: Stepsize

Require: $\beta,\ \gamma \in [0,\ 1)]$: Exponential decay rates for moment estimation

Require: $J(W)$: Cost function with parameters W

Require: W_0: Initial parameter

$m_0 \leftarrow 0$ (Initialize 1st moment vector)

$v_0 \leftarrow 0$ (Initialize 2nd moment vector)

$t \leftarrow 0$ (Initialize time step)

while W_t not converged **do**

$t \leftarrow t + 1$

$g_t \leftarrow \nabla J(W_{t-1})$(Get gradients w.r.t. cost function)

$m_t \leftarrow \beta m_{t-1} + (1-\beta) g_t$ (Update biased first moment estimate)

$v_t \leftarrow max(\gamma v_{t-1},\ |g_t|)$ (Update the exponentially weighted infinity norm)

$\hat{m}_t \leftarrow \frac{m_{t-1}}{1-\beta^t}$ (Compute bias-corrected first moment estimate)

$W_t \leftarrow W_{t-1} - \alpha \frac{\hat{m}_t}{v_t}$ (Update parameters)

end while

return W_t (Resulting parameters)

4.3.8 NAdam

NAG in Subsubsection 4.3.2 shows a powerful correction for momentum, and it is generally acknowledged that standard momentum is inferior to NAG. Dozat [22] improves Adam's momentum part to NAG and achieves faster convergence speed.

First, Dozat modifies the NAG by applying a look ahead momentum to the update:

$$\begin{aligned} v_t &= \beta v_{t-1} - \nabla J(W_{t-1}),\ v_0 = 0 \\ W_t &= W_{t-1} + \beta v_t + \alpha \nabla J(W_{t-1}) \end{aligned} \tag{4-43}$$

Then we can modify Adam in the following way:

$$\begin{aligned} W_t &= W_{t-1} - \alpha \frac{m_t}{(\sqrt{\hat{v}_t} + \epsilon)(1-\beta^t)} \\ &= W_{t-1} - \alpha \frac{\beta m_t + (1-\beta) g_t}{(\sqrt{\hat{v}_t} + \epsilon)(1-\beta^t)} \\ &= W_{t-1} - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon} (\beta \hat{m}_t + \frac{(1-\beta) g_t}{1-\beta^t}) \end{aligned} \tag{4-44}$$

A complete algorithm is shown in Algorithm 3.

Algorithm 3 NAdam. Note that all operators are element-wise. The recommended hyperparameters are learning rate $\alpha = 0.002$, momentum constant $\beta = 0.975$, RMSprop constant $\gamma = 0.999$, smoothing term $\epsilon = 10^{-8}$.

Require: α: Stepsize

Require: $\beta,\ \gamma \in [0,\ 1)]$: Exponential decay rates for moment estimation

Require: $J(W)$: Cost function with parameters W

Require: W_0: Initial parameter

$m_0 \leftarrow 0$ (Initialize 1st moment vector)

$v_0 \leftarrow 0$ (Initialize 2nd moment vector)

$t \leftarrow 0$ (Initialize time step)

while W_t not converged **do**

$t \leftarrow t + 1$

$g_t \leftarrow \nabla J(W_{t-1})$(Get gradients w.r.t. cost function)

$m_t \leftarrow \beta m_{t-1} + (1 - \beta) g_t$ (Update biased first moment estimate)

$v_t \leftarrow \gamma v_{t-1} + (1 - \gamma) g_t^2$ (Update biased second moment estimate)

$\hat{m}_t \leftarrow \frac{m_{t-1}}{1-\beta^t}$ (Compute bias-corrected first moment estimate)

$\hat{v}_t \leftarrow \frac{v_{t-1}}{1-\gamma^t}$ (Compute bias-corrected second raw moment estimate)

$W_t \leftarrow W_{t-1} - \frac{\alpha}{\sqrt{\hat{v}_t}+\epsilon}(\beta \hat{m}_t + \frac{(1-\beta) g_t}{1-\beta^t})$ (Update parameters)

end while

return W_t (Resulting parameters)

5 Main Questions in Neural Network Theory

Neural Networks have unfolded a striking power processing human like work. However, the mechanism underneath such success is beyond our comprehension. Center for Brains, Minds and Machines (CMBB) has done plentiful work on giving a satisfying characterization of the deep neural networks and completing our understanding of deep neural networks. In this section, we attend to three main questions put forward in Poggio et al. [23-25]: (1) Why deep neural networks avoid the exponentially increasing computation as the dimension increases comparing to shallow networks? (2) How does

the landscape of the empirical risk looks like in the deep neural networks?

5.1 Representation Power of Deep Neural Networks

The main conclusion that Poggio et al. [23] draw is that for all compositional functions with binary tree architecture deep neural networks eliminate the dependency on variable dimensionality in complexity.

5.1.1 Definition and Notation

Degree of approximation Let V_N be a set of all networks with complexity N[14], f be the target function and P be the approximation given by networks. Then the degree of approximation is:

$$\text{dist}(f,\ V_N) = \inf_{P \in V_N} ||f - P|| \tag{5-1}$$

If $\text{dist}(f,\ V_N) = O(N^{\gamma})$, then the complexity $N = O(\epsilon^{-\frac{1}{\gamma}})$ can guarantee a approximation with accuracy at least ϵ.

Function set Let $m,\ n \geq 1,\ m \in Z$, then W_m^n is the set of all functions of n variables with continuous partial derivatives of orders up to m. $W_m^{n,\ 2} \subseteq W_m^n$ is a set of all compositional functions of n variables with a binary tree architecture i.e. using functions $h \in W_m^2$. Let $k \in R^+$ is the degree of a polynomial, then P_k^n is the set of all polynomial functions of degree k in n variables. T_k^n is the set of all composition polynomials functions of degree k in n variables with a binary tree structure.

5.1.2 Shallow Networks

A theorem w.r.t. complexity of shallow network is stated in Mhaskar [26].

Theorem 5.1. *Let activation σ be infinitely differentiable and not a polynomial. For $f \in W_m^n$, the complexity of shallow networks that provide accuracy at least ϵ is:*

$$N = O(\epsilon^{-\frac{n}{m}}) \tag{5-2}$$

[14]The complexity we mean here is the total number of units in neural networks

The term $\epsilon^{-\frac{n}{m}}$ shows the exponentially increasing of neural network units in shallow networks.

5.1.3 Deep Hierarchy Networks

The second theorem is formulated in the binary tree structure see Figure 5.1. In CNN such structure corresponds to the 2×2 kernel.

Theorem 5.2. *Let activation σ be infinitely differentiable and not a polynomial. For $f \in W_m^{n,\,2}$, the complexity of shallow networks that provide accuracy at least ϵ is:*

$$N = O((n-1)\epsilon^{-\frac{2}{m}}) \tag{5-3}$$

Proof. We apply Theorem 5.1 to $h \in W_m^2$, which means the approximation provides accuracy $\epsilon = c_1 N^{-\frac{m}{2}},\ c_1 \in R$. We illustrate the proof using a example of $f(h_1,\ h_2),\ P(P_1,\ P_2)$. Since

$$\begin{aligned} ||f - P|| &\leq \epsilon \\ ||h_1 - P_1|| &\leq \epsilon \\ ||h_2 - P_2|| &\leq \epsilon \end{aligned} \tag{5-4}$$

Then we can derive:

$$||f(h_1,\ h_2) - P(P_1,\ P_2)|| = ||f(h_1,\ h_2) - f(P_1,\ P_2) + f(P_1,\ P_2) - P(P_1,\ P_2)|| \tag{5-5}$$

$$\leq ||f(h_1,\ h_2) - f(P_1,\ P_2)|| + ||f(P_1,\ P_2) - P(P_1,\ P_2)|| \tag{5-6}$$

$$\leq c_2 N^{-\frac{m}{2}} \tag{5-7}$$

Here we obtain the complexity of a single node in architecture, despite the output node, we come to the conclusion of $N = O((n-1)\epsilon^{-\frac{2}{m}})$.

Q.E.D.

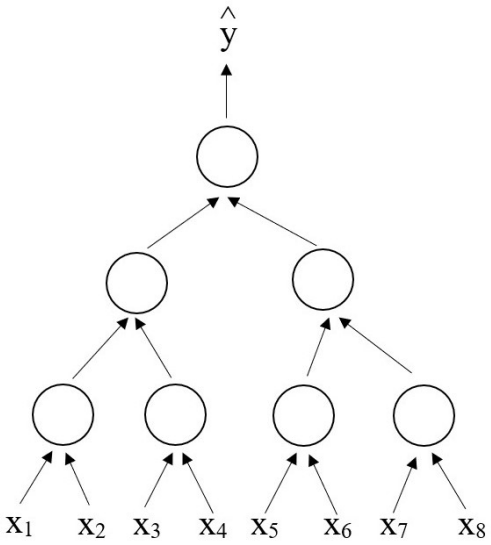


Figure 5.1: The binary tree structure

5.2 The Landscape of Empirical Risk

The term empirical risk is in contrast of the true risk, the average loss on the true distribution over all input and output. The true distribution is unknown, otherwise there is no need for neural networks. Using data set, subset of the whole distribution, as substitution is the origin of empirical. Poggio et al. [24] explore the landscape of the empirical risk, i.e. cost function J, under the framework of convolutional and over-parameterized neural networks and the activation functions $g^{[l]}(z)$ are polynomial-like ReLU[15]. Moreover, a landscape model satisfying the experimental results is provided.

First, we summarize the experimental observations in deep neural networks.

1. There is a large number of zero-error global minimums that are degenerate.[16]

2. Perturbation will lead to different convergence path, the larger or earlier in training process the perturbation is, the more different the path and the final minimum will be.

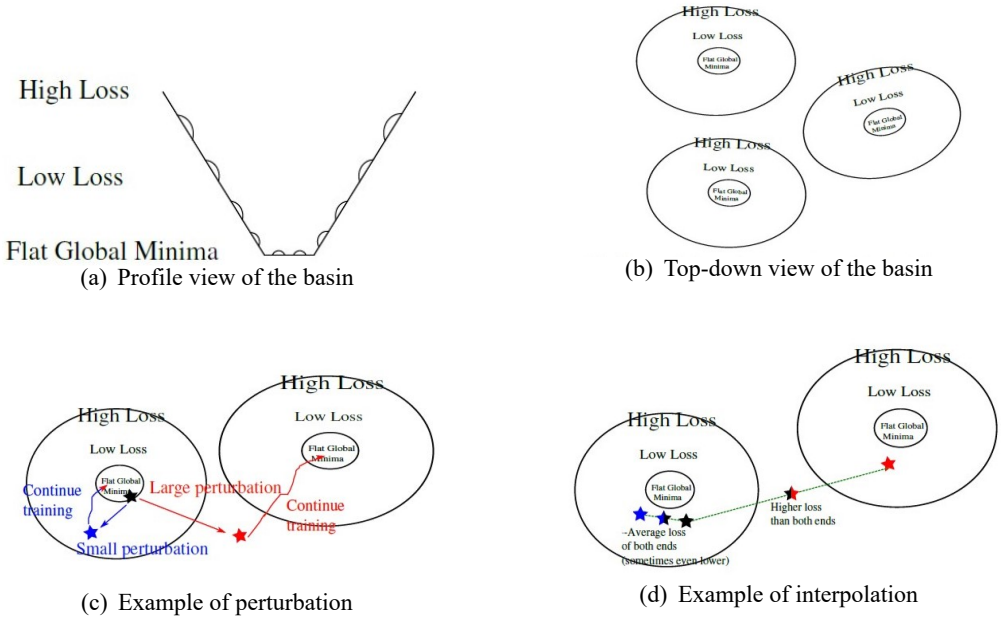
[15]The ReLU is approximated by polynomials whose degree is $d^{[l]}(\epsilon)$ at the accuracy of ϵ.

[16]See Appendix 3 for theoretical analysis

3. Interpolation between two close convergence path to the same minimum will have similar or even smaller error. On the other hand, interpolation between two minimums cause the rise of error.

4. No local minimum is observed.

A simple model that is consistent with the observations is a collection of rugged basins which is shown in Figure 5.2. The flat and wide global minimums in basins explain a myriad of zero-error minimizers. A small perturbation will not make a difference in a basin, while a lager one may cause a different path to other basins, see **??**. Besides, the ”walls” between basins demonstrated in **??** may be the cause of interpolation’s different behaviors in the neighbor of minimum and a distinct minimum.



(a) Profile view of the basin

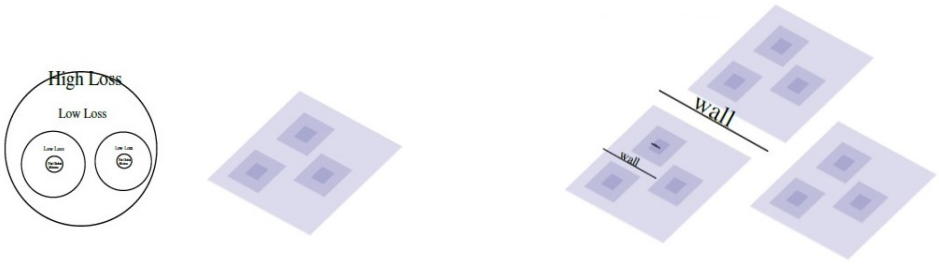
(b) Top-down view of the basin

(c) Example of perturbation

(d) Example of interpolation

Figure 5.2: The basin model

Another interesting model is a variant of basin: the basin-fractal Figure 5.3. The key difference between simple basins and basin-fractal is that there are more than one global minimum in the same low loss area in the latter model. Though in most cases the ”walls” between these global minimums are too flat and smooth to notice.



(a) 2D view of basin-fractal (b) The landscape of basin-fractal

Figure 5.3: The basin-fractal model

5.3 Generalization of Neural Networks

There is no use for neural networks only fit the training data well. The character of a neural network is measured by how well it performs on other data, which is how well it generalizes. Due to the obscurity of mathematical analysis in Banburski et al. [25], we only give a summary on the whole work.

1. Using the exponential loss[17] , minimization of J with a vanishing regularization term in weights leads to the maximization of the margin of f.

2. The critical points are hyperbolic equilibrium points[18] if the data are separable.

6 Conclusions

The goal of this article is to review the knowledge that I had learnt during the winter holiday and pave the way for further study in the future. First we formulated the framework of neural networks, introduced the basic component in network and the gradient descent optimization together with the computation method: backward propagation. Then we summarized the dominating problems, especially the vanishing/explosion and the saddle points, in training a neural network and provided a few tricks including LSUV initialization and Adam optimization that help boost the training efficiency. Finally, we

[17]The exponential loss is defined as $l(yf) = e^{-yf}$, then the cost function is $J(f) = \sum_{n=1}^{N} l\left(y_n f\left(x_n\right)\right)$

[18]A hyperbolic equilibrium point is a point at which the Jacobian Matrix has no eigenvalue with zero real part.[27]

went through the main three questions about the mechanism of neural networks and obtained some straight-forward intuition like the landscape comprised of flat basins.

There are several potential work to be done in the future. The tensorflow and keras framework can be used to compare the performance of various tricks and empirically test the results given in Section 5. Besides, due to lack of mathematic knowledge, the majority work in the generalization is beyond my comprehension. With further study of the basic convex optimization, a good command of mathematic terminology and definition may help me have more insight into the essence of neural networks generalization.

7 References

[1] DAUPHIN Y N, PASCANU R, GULCEHRE C, et al. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization[C]// Advances in neural information processing systems. [S.l. : s.n.], 2014: 2933-2941.

[2] BERTSEKAS D P. Nonlinear programming[J]. Journal of the Operational Research Society, 1997, 48(3): 334-334.

[3] SUN R. Optimization for deep learning: theory and algorithms[J]. ArXiv preprint arXiv:1912.08957, 2019.

[4] SHAMIR O. Exponential convergence time of gradient descent for one-dimensional deep linear neural networks[J]. ArXiv preprint arXiv:1809.08587, 2018.

[5] BRAY A J, DEAN D S. Statistics of critical points of Gaussian fields on large-dimensional spaces[J]. Physical review letters, 2007, 98(15): 150201.

[6] MISHKIN D, MATAS J. All you need is a good init[J]. ArXiv preprint arXiv:1511.06422, 2015.

[7] ROMERO A, BALLAS N, KAHOU S E, et al. Fitnets: Hints for thin deep nets[J]. ArXiv preprint arXiv:1412.6550, 2014.

[8] GLOROT X, BENGIO Y. Understanding the difficulty of training deep feedforward neural networks[C]//Proceedings of the thirteenth international conference on artificial intelligence and statistics. [S.l. : s.n.], 2010: 249-256.

[9] JIA Y, SHELHAMER E, DONAHUE J, et al. Caffe: Convolutional architecture for fast feature embedding[C]//Proceedings of the 22nd ACM international conference on Multimedia. [S.l. : s.n.], 2014: 675-678.

[10] HE K, ZHANG X, REN S, et al. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification[C]//Proceedings of the IEEE international conference on computer vision. [S.l. : s.n.], 2015: 1026-1034.

[11] SAXE A M, MCCLELLAND J L, GANGULI S. Exact solutions to the non-linear dynamics of learning in deep linear neural networks[J]. ArXiv preprint arXiv:1312.6120, 2013.

[12] LECUN Y. THE MNIST DATABASE of handwritten digits[Z]. http://yann.lecun.com/exdb/mnist/. Accessed Feburary 28, 2020.

[13] QIAN N. On the momentum term in gradient descent learning algorithms[J]. Neural networks, 1999, 12(1): 145-151.

[14] ORR G B. Momentum and Learning Rate Adaptation[Z]. https://www.willamette.edu/~gorr/classes/cs449/momrate.html. Accessed Feburary 20, 2020.

[15] BURDEN R L, FAIRES D J. Numerical analysis[J]., 2010.

[16] NESTEROV Y. A method for unconstrained convex minimization problem with the rate of convergence O (1/k^ 2)[C]//Doklady an ussr:vol. 269. [S.l. : s.n.], 1983: 543-547.

[17] SUTSKEVER I. Training recurrent neural networks[M]. [S.l.]: University of Toronto Toronto, Ontario, Canada, 2013.

[18] HINTON G. Overview of Mini-batch Gradient Descent[Z]. http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf. Accessed Feburary 21, 2020.

[19] DUCHI J, HAZAN E, SINGER Y. Adaptive subgradient methods for online learning and stochastic optimization[J]. Journal of machine learning research, 2011, 12(Jul): 2121-2159.

[20] ZEILER M D. Adadelta: an adaptive learning rate method[J]. ArXiv preprint arXiv:1212.5701, 2012.

[21] KINGMA D P, BA J. Adam: A method for stochastic optimization[J]. ArXiv preprint arXiv:1412.6980, 2014.

[22] DOZAT T. Incorporating nesterov momentum into adam[J]., 2016.

[23] POGGIO T, MHASKAR H, ROSASCO L, et al. Theory i: Why and when can deep networks avoid the curse of dimensionality?[R]. [S.l.]: Center for Brains, Minds, 2016.

[24] POGGIO T, LIAO Q. Theory II: Landscape of the empirical risk in deep learning[D]. Center for Brains, Minds, 2017.

[25] BANBURSKI A, LIAO Q, MIRANDA B, et al. Theory III: Dynamics and generalization in deep networks[J]. ArXiv preprint arXiv:1903.04991, 2019.

[26] MHASKAR H N. Neural networks for optimal approximation of smooth and analytic functions[J]. Neural computation, 1996, 8(1): 164-177.

[27] Wikipedia. Hyperbolic Equilibrium Point[Z]. https://wikivisually.com/wiki/Hyperbolic_equilibrium_point. Accessed Feburary 23, 2020.

Appendix

1 Descent Lemma

Lemma 1.1. *Let $f : \mathbb{R}^n \to R$ be continuous differentiable, and $x,\ y \in \mathbb{R}^n$. Suppose that:*

$$\forall t \in [0,\ 1]\ \ \|\nabla f(x+ty) - \nabla f(x)\| \leq Lt\|y\| \tag{1-1}$$

where L is a scalar, then:

$$f(x+y) \leq f(x) + y^T \nabla f(x) + \frac{L}{2}\|y\|^2 \tag{1-2}$$

Proof. Let $g(t) = f(x+ty)$, then:

$$\begin{aligned}
f(x+y) - f(x) &= g(1) - g(0) && (1\text{-}3)\\
&= \int_0^1 \frac{dg}{dt}(t)dt && (1\text{-}4)\\
&= \int_0^1 y^T \nabla f(x+ty)dt && (1\text{-}5)\\
&\leq \int_0^1 y^T \nabla f(x)dt + \left| \int_0^1 y^T (\nabla f(x+ty) - \nabla f(x))dt \right| && (1\text{-}6)\\
&\leq \int_0^1 y^T \nabla f(x)dt + \int_0^1 \|y\| \cdot \|\nabla f(x+ty) - \nabla f(x)\|dt && (1\text{-}7)\\
&\leq y^T \nabla f(x) + \|y\| \int_0^1 Lt\|y\|dt && (1\text{-}8)\\
&= y^T \nabla f(x) + \frac{L}{2}\|y\|^2 && (1\text{-}9)
\end{aligned}$$

2 PReLU

He et al. [10] first propose the variant PReLU, The definition is:

$$g\left(z_j\right) = \begin{cases} z_j, & \text{if } z_j > 0 \\ a_j z_j, & \text{if } z_j \leq 0 \end{cases} \tag{2-1}$$

or in a more compact form:

$$g(z_j) = \max\left(0, z_j\right) + a_j \min\left(0, z_j\right) \tag{2-2}$$

a_j is the coefficient that can be adaptively learnt, subscript j means the coefficient can vary on the different channel in the same layer. Though the shape of PReLU and Leaky ReLU (LReLU) look the same, PReLU drastically avoids the overfitting by introducing a few parameters. In practice, if a_j is shared by the same layers the performance will be better.

3 Numerous Zero-error Minimum

Theorem 3.1. *Suppose $\mathbb{F}$ is a field and polynomials $P,\ Q \in \mathbb{F}$ have no common factor of degree $d \geq 1$* [19]*. Let $Z(P,\ Q) = \{(x,\ y) \in \mathbb{F}^2 | P(x,\ y) = Q(x,\ y)\}$. Then the number of points is upper bounded by $d(P)d(Q)$.*

Via Bezout theorem Theorem 3.1, we can obtain the upper bound of a system with l layers and N data: d^{lN}, where d is the degree of polynomial equations. In the language of neural networks, the number of zero points of cost functions is d^{lN}. The theorem can not strictly apply to neural networks with ReLU. But experimental in Figure 2 in [24] results show the similar behavior of ReLU and polynomial approximation. Therefore, we give a proposition as follow:

Proposition 3.1. *There are a large number of zero-error minimum.*

[19] A more frank expression is independent.

Acknowledgements

Being a undergraduate student in Shannon Class is lots of fun, and the task in winter holiday greatly enriched my knowledge. I want to thank my tutor, Yu Guanding. Yu brought me into his academic group and revealed me the goal to fight for. He is a terrific tutor who gives me freedom to make decisions and highlights the academic route in the future.

It is a fortune to meet graduate student Liu Wenliang. As a senior, he provided me sufficient support on getting familiar with neural networks. I want to thank Liu for his unfailing help whenever I have some questions.

I want to thank the current learning student Zhou Kaining and Li Jiatong for giving me suggestions on the choice of neural networks courses, and stimulate me to exert all my power to study and research during the holiday.

But most of all, I want to express my gratitude to my family. During the burst of NCP, my parents did a lot to keep the house clean and cozy. Thank you for your support.