Backpropagation and Optimization in Deep Learning: Tutorial and Survey

Benyamin Ghojogh, Ali Ghodsi

Backpropagation and Optimization in Deep Learning: Tutorial and Survey

Benyamin Ghojogh, Ali Ghodsi {BGHOJOGH, ALI.GHODSI}@UWATERLOO.CA
Waterloo, Ontario, Canada

Abstract

This is a tutorial and survey paper on backpropagation and optimization in neural networks. It starts with gradient descent, line-search, momentum, and steepest descent. Then, backpropagation is introduced. Afterwards, stochastic gradient descent, mini-batch stochastic gradient descent, and their convergence rates are discussed. Adaptive learning rate methods, including AdaGrad, RMSProp, and Adam, are explained. Then, algorithms for sharpness-aware minimization are introduced. Finally, convergence guarantees for optimization in over-parameterized neural networks are discussed.

1. Introduction

Machine learning is nothing but optimization plus some other tools such as linear algebra, probability, and statistics. Deep learning, as a family of machine learning algorithms, is also merely optimization with novelty on devising practical loss functions. In fact, deep learning minimizes some loss function with optimization algorithms. Different optimization algorithms for training neural networks have been proposed, some of which are backpropagation (Rumelhart et al., 1986), genetic algorithms (Montana & Davis, 1989; Leung et al., 2003), and belief propagation as in restricted Boltzmann machines (Hinton & Salakhutdinov, 2006).

The most well-known and widely used optimization algorithm for deep learning is backpropagation. Backpropagation is a combination of gradient descent and chain rule in derivatives. It is possible to use second-order optimization, such as Newton's method (Ghojogh et al., 2023a), for training neural networks; however, first-order optimization, i.e., gradient descent, is usually used in deep learning. This is because calculation of Hessian in second-order optimization is not efficient and the fast first-order optimization have been found to be sufficient for training neural networks.

This chapter discusses backpropagation and optimization in deep learning. It starts with gradient descent, line-search, momentum, and steepest descent. Then, backpropagation is introduced. Afterwards, stochastic gradient descent, mini-batch stochastic gradient descent, and their convergence rates are discussed. Adaptive learning rate methods, including AdaGrad, RMSProp, and Adam, are explained. Then, algorithms for sharpness-aware minimization are introduced. Finally, convergence guarantees for optimization in over-parameterized neural networks are discussed.

2. Gradient Descent

2.1. Gradient Descent

Backpropagation and many optimization algorithms in neural networks are usually first-order optimization methods. The most well-known first-order optimization algorithm is gradient descent. In fact, as it will be explained in this chapter, backpropagation is nothing but gradient descent and chain rule in derivatives. Therefore, we start with gradient descent. Gradient descent is one of the fundamental first-order methods. It was first suggested by Cauchy in 1874 (Lemaréchal, 2012) and Hadamard in 1908 (Hadamard, 1908) and its convergence was later analyzed in (Curry, 1944).

Consider the following unconstrained optimization of the function $f(.)$ with respect to the variable $\boldsymbol{w}$:

$$\underset{\boldsymbol{w}}{\text{minimize}} \quad f(\boldsymbol{w}). \tag{1}$$

Numerical optimization for unconstrained optimization starts with a random feasible initial point and iteratively updates it by step $\Delta\boldsymbol{w}$:

$$\boldsymbol{w}^{(k+1)} := \boldsymbol{w}^{(k)} + \Delta\boldsymbol{w}. \tag{2}$$

It continues until convergence to (or getting sufficiently close to) the desired optimal point $\boldsymbol{w}^*$.

Assume the gradient of function $f(\boldsymbol{w})$ is L-smooth where L is the Lipschitz constant[1]. In gradient descent, the update at every iteration is (see (Ghojogh et al., 2021; 2023a) for

[1]The function $f(.)$ is Lipschitz with Lipschitz constant L if $|f(\boldsymbol{w}_1) - f(\boldsymbol{w}_2)| \leq L \|\boldsymbol{w}_1 - \boldsymbol{w}_2\|_2, \forall \boldsymbol{w}_1, \boldsymbol{w}_2 \in \mathcal{D}$. In other words, the Lipschitz constant can be seen as an upper bound on the slope of function in its domain $\mathcal{D}$. Note that here, the gradient of function is assumed to be Lipschitz smooth meaning that L is an upper bound on the slope of gradient of function, i.e., $|\nabla f(\boldsymbol{w}_1) - \nabla f(\boldsymbol{w}_2)| \leq L \|\boldsymbol{w}_1 - \boldsymbol{w}_2\|_2, \forall \boldsymbol{w}_1, \boldsymbol{w}_2 \in \mathcal{D}$

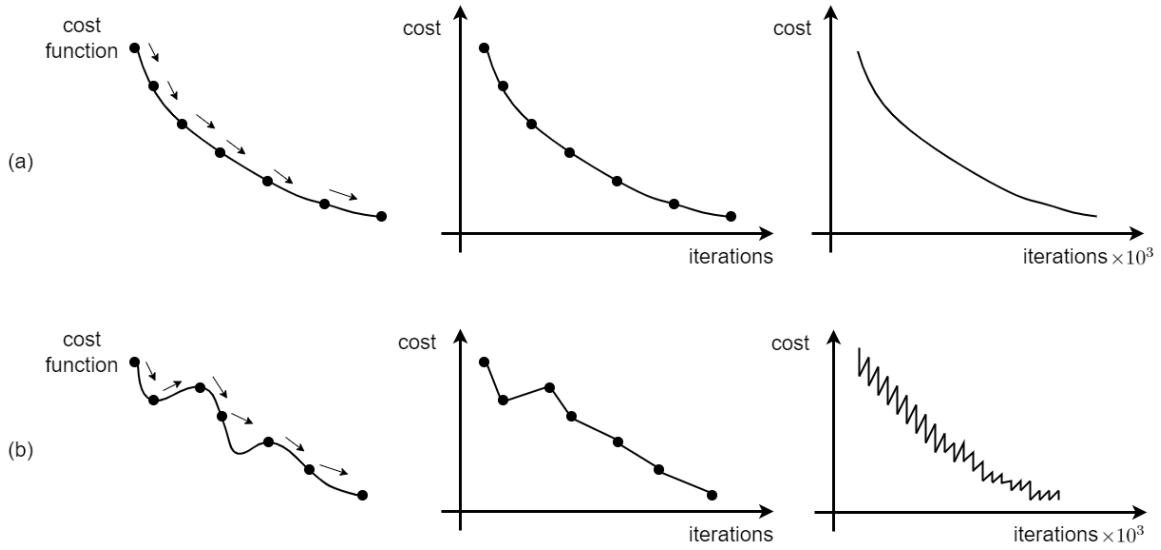

Figure 1. The gradient descent steps for (a) a convex cost function and (b) a non-convex cost function. The left to right figures depict the optimization steps on cost function, the cost value at the iterations of optimization, and the cost value in large number of iterations.

proof):

$$\Delta \boldsymbol{w} = -\frac{1}{L}\nabla f(\boldsymbol{w}^{(k)}) \implies \boldsymbol{w}^{(k+1)} := \boldsymbol{w}^{(k)} - \frac{1}{L}\nabla f(\boldsymbol{w}^{(k)}). \tag{3}$$

The problem is that either the Lipschitz constant L is often not known or it is hard to compute. Hence, rather than $\Delta \boldsymbol{w} = -\frac{1}{L}\nabla f(\boldsymbol{w}^{(k)})$, we use:

$$\Delta \boldsymbol{w} = -\eta \nabla f(\boldsymbol{w}^{(k)}), \text{ i.e., } \boldsymbol{w}^{(k+1)} := \boldsymbol{w}^{(k)} - \eta \nabla f(\boldsymbol{w}^{(k)}), \tag{4}$$

where $\eta > 0$ is the step size, also called the learning rate in data science literature. If the optimization problem is maximization rather than minimization, the step should be $\Delta \boldsymbol{w} = \eta \nabla f(\boldsymbol{w}^{(k)})$ rather than Eq. (4). In that case, the name of method is gradient ascent. The learning rate can be found by line search which is often used in optimization and not in deep learning. Line search will be discussed in Section 2.3.

For a convex function, the series of solutions converges to the optimal solution while the function value decreases iteratively until the local minimum:

$$\{\boldsymbol{w}^{(0)}, \boldsymbol{w}^{(1)}, \boldsymbol{w}^{(2)}, \dots\} \to \boldsymbol{w}^*,$$
$$f(\boldsymbol{w}^{(0)}) \geq f(\boldsymbol{w}^{(1)}) \geq f(\boldsymbol{w}^{(2)}) \geq \cdots \geq f(\boldsymbol{w}^*).$$

If the optimization problem is a convex problem, the solution is the global solution; otherwise, the solution is local. As Fig. 1 illustrates, gradient descent even works relatively fine for non-convex cost functions; it might oscillate for non-convex cost but its overall pattern is decreasing.

2.2. Convergence Criteria

For all numerical optimization methods including gradient descent, there exist several methods for convergence criterion to stop updating the solution and terminate optimization. Some of them are:

- Small norm of gradient:

$$\|\nabla f(\boldsymbol{w}^{(k+1)})\|_2 \leq \epsilon,$$

  where ϵ is a small positive number. The reason for this criterion is the first-order optimality condition, stating that at the local optimum, there is $\|\nabla f(\boldsymbol{w}^*)\|_2 = 0$. If the function is not convex, this criterion has the risk of stopping at a saddle point.

- Small change of cost function:

$$|f(\boldsymbol{w}^{(k+1)}) - f(\boldsymbol{w}^{(k)})| \leq \epsilon.$$

- Small change of gradient of function:

$$|\nabla f(\boldsymbol{w}^{(k+1)}) - \nabla f(\boldsymbol{w}^{(k)})| \leq \epsilon.$$

- Reaching maximum desired number of iterations.

2.3. Line-search

It was explained that the step size of gradient descent requires knowledge of the Lipschitz constant for the smoothness of gradient. However, the exact Lipschitz constant may not be known, especially that it is usually hard to compute. Alternatively, the suitable step size η can be found by a search which is named the line-search. The line-search of

Initialize $\boldsymbol{w}^{(0)}$
for *iteration* $k = 0, 1, \dots$ **do**
  Initialize $\eta := 1$
  for *iteration* $\tau = 1, 2, \dots$ **do**
    Check line-search condition, i.e., Eq. (6)
    if *not satisfied* **then**
      $\eta \leftarrow \frac{1}{2} \times \eta$
    else
      $\boldsymbol{w}^{(k+1)} := \boldsymbol{w}^{(k)} - \eta \nabla f(\boldsymbol{w}^{(k)})$
      break the loop
  Check the convergence criterion
  if *converged* **then**
    return $\boldsymbol{w}^{(k+1)}$

Algorithm 1: Gradient descent with line search

every optimization iteration starts with $\eta = 1$ and if it does not satisfy:

$$f(\boldsymbol{w}^{(k)} + \Delta \boldsymbol{w}) - f(\boldsymbol{w}^{(k)}) < 0, \tag{5}$$

with step $\Delta \boldsymbol{w} = -\eta \nabla f(\boldsymbol{w}^{(k)})$:

$$\begin{aligned} & f(\boldsymbol{w}^{(k)} + \Delta \boldsymbol{w}) < f(\boldsymbol{w}^{(k)}) \\ & \implies f(\boldsymbol{w}^{(k)} - \eta \nabla f(\boldsymbol{w}^{(k)})) < f(\boldsymbol{w}^{(k)}), \end{aligned} \tag{6}$$

the step size is halved, $\eta \leftarrow \eta/2$. This halving step size is repeated until this equation is satisfied, i.e., until there is a decrease in the objective function. Note that this decrease will happen when the step size becomes small enough to satisfy (see (Ghojogh et al., 2021; 2023a) for proof):

$$\eta < \frac{1}{L}. \tag{7}$$

A more sophisticated line-search method is the Armijo line-search (Armijo, 1966), also called the backtracking line-search. Another more sophisticated line-search is Wolfe conditions (Wolfe, 1969). More details of these can be studied in (Ghojogh et al., 2021; 2023a). The algorithm of gradient descent with line-search is Algorithm 1. As this algorithm shows, line-search has its own internal iterations inside every iteration of gradient descent.

2.4. Momentum

Gradient descent and other first-order methods can have a momentum term. Momentum, proposed in (Rumelhart et al., 1986), makes the change of solution $\Delta \boldsymbol{w}$ a little similar to the previous change of solution. Therefore, the change adds a history of previous change to Eq. (4):

$$(\Delta \boldsymbol{w})^{(k)} := \alpha (\Delta \boldsymbol{w})^{(k-1)} - \eta^{(k)} \nabla f(\boldsymbol{w}^{(k)}), \tag{8}$$

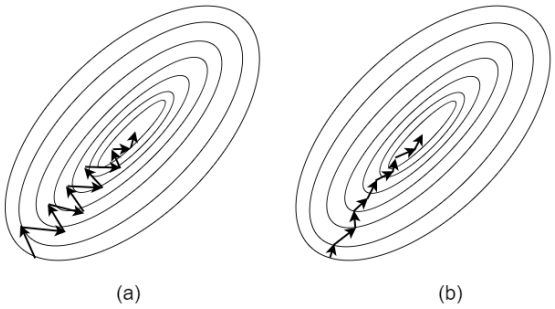

Figure 2. Gradient descent (a) without and (b) with momentum. Each contour shows the same cost value in the optimization. As the figure shows, momentum reduces oscillation of optimization.

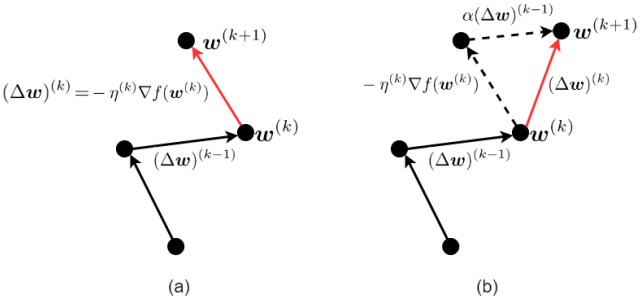

Figure 3. Update of solution in gradient descent (a) without and (b) with momentum according to Eq. (8). According to the addition of terms in this equation, oscillation is reduced.

where $\alpha > 0$ is the momentum parameter which weights the importance of history compared to the descent direction. We use this $(\Delta \boldsymbol{w})^{(k)}$ in Eq. (2) for updating the solution:

$$\boldsymbol{w}^{(k+1)} := \boldsymbol{w}^{(k)} + (\Delta \boldsymbol{w})^{(k)}.$$

Because of faithfulness to the track of previous updates, momentum reduces the amount of oscillation of updates in gradient descent optimization. This effect is shown in Fig. 2. Moreover, the addition of terms in Eq. (8) is illustrated in Fig. 3 explains mathematically how the reduction of oscillation works when using the momentum term.

2.5. Steepest Descent

Steepest descent is similar to gradient descent but there is a difference between them. In steepest descent, the solution moves toward the negative gradient as much as possible to reach the smallest function value which can be achieved at every iteration. Hence, the step size at iteration k of steepest descent is calculated as (Chong & Zak, 2004):

$$\eta^{(k)} := \arg\min_{\eta} \ f\big(\boldsymbol{w}^{(k)} - \eta \nabla f(\boldsymbol{w}^{(k)})\big), \tag{9}$$

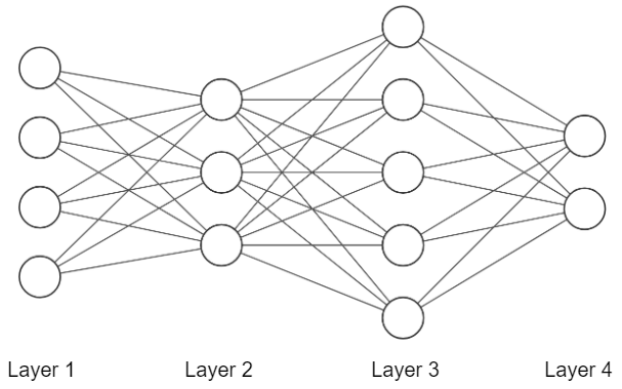

Figure 4. A feed-forward neural network with four layers.

and then, the solution is updated using Eq. (4) as in gradient descent:

$$\boldsymbol{w}^{(k+1)} := \boldsymbol{w}^{(k)} - \eta \nabla f(\boldsymbol{w}^{(k)}).$$

3. Backpropagation

Consider a feed-forward neural network with four layers depicted in Fig. 4. Note that, depending on whether to consider weights or nodes as layers in neural network, one might call the network of this figure have either three or four layers, respectively. Here, for the sake of explanation of backpropagation, the nodes are considered to be layers in the network.

Every neuron i in the neural network is depicted in Fig. 5. Let w_{ji} denote the weight connecting neuron i to neuron j. Let a_i and z_i be the output of neuron i before and after applying its activation function $\sigma_i(.) : \mathbb{R} \to \mathbb{R}$, respectively:

$$a_i = \sum_{\ell=1}^{m} w_{i\ell} z_\ell, \tag{10}$$

$$z_i := \sigma_i(a_i). \tag{11}$$

Consider three neurons in three successive layers of a network as illustrated in Fig. 6. Consider Eq. (10), i.e., $a_i = \sum_\ell w_{i\ell} z_\ell$, which sums over the neurons in layer ℓ. By chain rule in derivatives, the gradient of error e with respect to to the weight between neurons ℓ and i is:

$$\frac{\partial e}{\partial w_{i\ell}} = \frac{\partial e}{\partial a_i} \times \frac{\partial a_i}{\partial w_{i\ell}} \overset{(a)}{=} \delta_i \times z_\ell, \tag{12}$$

where (a) is because $a_i = \sum_\ell w_{i\ell} z_\ell$ and we define:

$$\delta_i := \frac{\partial e}{\partial a_i}. \tag{13}$$

If layer i is the last layer, δ_i can be computed by derivative of error (loss function) with respect to the output. However,

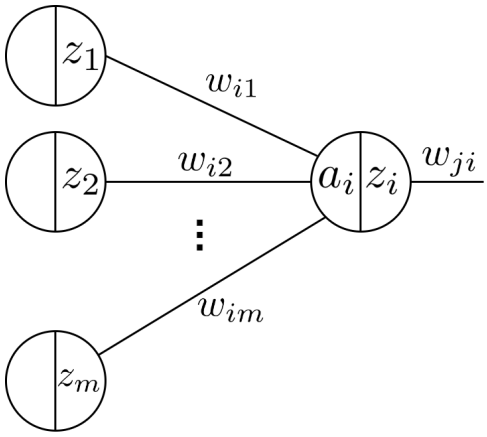

Figure 5. Neuron i in the neural network.

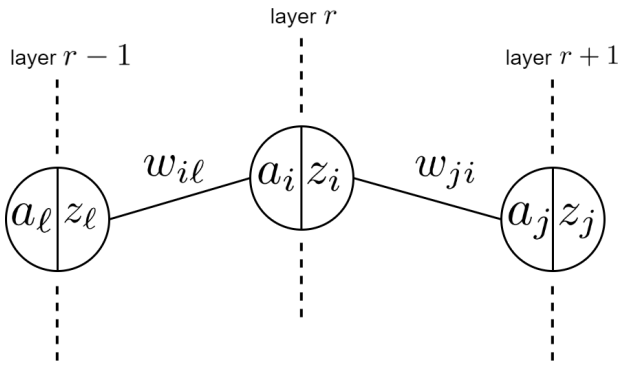

Figure 6. Three neurons in three successive layers of a network.

if i is one of the hidden layers, δ_i is computed by chain rule as:

$$\delta_i = \frac{\partial e}{\partial a_i} = \sum_j \Big(\frac{\partial e}{\partial a_j} \times \frac{\partial a_j}{\partial a_i} \Big) = \sum_j \Big(\delta_j \times \frac{\partial a_j}{\partial a_i} \Big). \tag{14}$$

The term $\partial a_j / \partial a_i$ is calculated by chain rule as:

$$\frac{\partial a_j}{\partial a_i} = \frac{\partial a_j}{\partial z_i} \times \frac{\partial z_i}{\partial a_i} \overset{(a)}{=} w_{ji}\, \sigma'(a_i), \tag{15}$$

where (a) is because $a_j = \sum_i w_{ji} z_i$ and $z_i = \sigma(a_i)$ and $\sigma'(.)$ denotes the derivative of activation function. Putting Eq. (15) in Eq. (14) gives:

$$\delta_i = \sigma'(a_i) \sum_j (\delta_j\, w_{ji}). \tag{16}$$

Putting this equation in Eq. (12) gives:

$$\frac{\partial e}{\partial w_{i\ell}} = z_\ell\, \sigma'(a_i) \sum_j (\delta_j\, w_{ji}). \tag{17}$$

Initialize $\boldsymbol{w}^{(0)}$
for *iteration* $k = 0, 1, \ldots$ **do**
  Initialize the learning rate $\eta^{(0)}$
  for *layer* r *from the last layer to the first layer* **do**
    for *neuron* i *in the layer* r **do**
      for *neuron* ℓ *in the layer* $(r-1)$ **do**
        if *layer* r *is the last layer* **then**
          $w_{i\ell}^{(k+1)} := w_{i\ell}^{(k)} - \eta^{(k)} z_\ell \frac{\partial e}{\partial a_i}$
        else
          $w_{i\ell}^{(k+1)} := w_{i\ell}^{(k)} - \eta^{(k)} z_\ell \, \sigma'(a_i) \sum_j (\delta_j \, w_{ji})$
  Check the convergence criterion
  if *converged* **then**
    return all weights of neural network
  Adapt (update) the learning rate $\eta^{(k)}$.

Algorithm 2: Backpropagation algorithm

Backpropagation uses the gradient in Eq. (17) for updating the weight $w_{i\ell}, \forall i, \ell$ by gradient descent:

$$w_{i\ell}^{(k+1)} := w_{i\ell}^{(k)} - \eta^{(k)} \frac{\partial e}{\partial w_{i\ell}}, \quad \forall i, \ell. \tag{18}$$

Therefore, for the weights of the last layer (if i denotes the neurons in the last layer), the gradient descent becomes:

$$w_{i\ell}^{(k+1)} := w_{i\ell}^{(k)} - \eta^{(k)} z_\ell \frac{\partial e}{\partial a_i}, \quad \forall i, \ell, \tag{19}$$

according to Eqs. (12) and (13). For the weights of the other layers (if i denotes the neurons in a hidden layer), the gradient descent becomes:

$$w_{i\ell}^{(k+1)} := w_{i\ell}^{(k)} - \eta^{(k)} z_\ell \, \sigma'(a_i) \sum_j (\delta_j \, w_{ji}), \quad \forall i, \ell, \tag{20}$$

according to Eq. (17).

The backpropagation algorithm is in Algorithm 2. This tunes the weights from last layer to the first layer for every iteration of optimization. Therefore, backpropagation, proposed in 1986 (Rumelhart et al., 1986), is actually gradient descent with chain rule in derivatives because of having layers of parameters (so that the gradients for weights of every layer depend on the the gradients for weights of later layers in the network). It is the most well-known optimization method used for training neural networks.

4. Stochastic Gradient Descent

4.1. Algorithm of Stochastic Gradient Descent

Assume there is a dataset of n data instances, $\{\boldsymbol{x}_i \in \mathbb{R}^d\}_{i=1}^n$ and their labels $\{l_i \in \mathbb{R}\}_{i=1}^n$. Let the cost function $f(.)$ be decomposed into summation of n terms $\{f_i(\boldsymbol{w})\}_{i=1}^n$. In other words, the neural network has a loss value $f_i(\boldsymbol{w})$ per every input data instance $\boldsymbol{x}_i$. Therefore, the total loss is the average of loss values over the n data instances and the optimization problem becomes:

$$\underset{\boldsymbol{w}}{\text{minimize}} \quad \frac{1}{n} \sum_{i=1}^n f_i(\boldsymbol{w}). \tag{21}$$

In this case, the full gradient is the average gradient, i.e:

$$\nabla f(\boldsymbol{w}) = \frac{1}{n} \sum_{i=1}^n \nabla f_i(\boldsymbol{w}), \tag{22}$$

so the update in gradient descent, i.e., Eq. (3), becomes:

$$\Delta \boldsymbol{w} = -\eta \nabla f(\boldsymbol{w}^{(k)}) = -\frac{\eta}{n} \sum_{i=1}^n \nabla f_i(\boldsymbol{w}^{(k)}).$$

This is what gradient descent uses for updating the solution at every iteration.

Calculation of the full gradient is time-consuming and inefficient for large values of n, especially as it needs to be recalculated at every iteration of gradient descent. Stochastic Gradient Descent (SGD), also called stochastic gradient method, approximates gradient descent stochastically and samples (i.e. bootstraps) one of the points at every iteration for updating the solution. Therefore, it uses:

$$\boldsymbol{w}^{(k+1)} := \boldsymbol{w}^{(k)} - \eta^{(k)} \nabla f_i(\boldsymbol{w}^{(k)}), \tag{23}$$

rather than Eq. (4), $\boldsymbol{w}^{(k+1)} := \boldsymbol{w}^{(k)} - \eta \nabla f(\boldsymbol{w}^{(k)})$. The idea of stochastic approximation was first proposed in 1951 (Robbins & Monro, 1951). It was first used for machine learning in 1998 (Bottou et al., 1998).

As Eq. (23) states, SGD often uses an adaptive step size which changes in every iteration. The step size can be decreasing because in initial iterations, where the solution is far away from the optimal solution, the step size can be large; however, it should be small in the last iterations which is supposed to be close to the optimal solution. Some well-known adaptations for the step size are:

$$\eta^{(k)} := \frac{1}{k},$$
$$\eta^{(k)} := \frac{1}{\sqrt{k}},$$
$$\eta^{(k)} := \eta.$$

4.2. Convergence Analysis of Stochastic Gradient Descent

Proposition 1 (Convergence rate of gradient descent with full gradient)**.** *Consider a convex and differentiable function $f(.)$, with domain $\mathcal{D}$, whose gradient is L-smooth. Let f^* be the minimum of cost function and $\boldsymbol{w}^*$ be the minimizer. Starting from the initial solution $\boldsymbol{w}^{(0)}$, after t iterations of the optimization algorithm, the convergence rate of gradient descent is:*

$$f(\boldsymbol{w}^{(t+1)}) - f^* \leq \frac{2L\|\boldsymbol{w}^{(0)} - \boldsymbol{w}^*\|_2^2}{t+1} = \mathcal{O}(\frac{1}{t}). \quad (24)$$

Proposition 2 (Convergence rate of gradient descent with full gradient)**.** *Consider a function $f(\boldsymbol{w}) = \sum_{i=1}^n f_i(\boldsymbol{w})$ and which is bounded below and each f_i is differentiable. Let the domain of function $f(.)$ be $\mathcal{D}$ and its gradient be L-smooth. Assume $\mathbb{E}[\|\nabla f_i(\boldsymbol{w}_k)\|_2^2 \,|\, \boldsymbol{w}_k] \leq \beta^2$ where β is a constant. Assume $\mathbb{E}[\|\nabla f_i(\boldsymbol{w}_k)\|_2^2 \,|\, \boldsymbol{w}_k] \leq \beta^2$ where β is a constant. Depending on the step size, the convergence rate of SGD is:*

$$f(\boldsymbol{w}^{(t+1)}) - f^* \leq \mathcal{O}(\frac{1}{\log t}) \quad \textit{if} \quad \eta^{(\tau)} = \frac{1}{\tau}, \quad (25)$$

$$f(\boldsymbol{w}^{(t+1)}) - f^* \leq \mathcal{O}(\frac{\log t}{\sqrt{t}}) \quad \textit{if} \quad \eta^{(\tau)} = \frac{1}{\sqrt{\tau}}, \quad (26)$$

$$f(\boldsymbol{w}^{(t+1)}) - f^* \leq \mathcal{O}(\frac{1}{t} + \eta) \quad \textit{if} \quad \eta^{(\tau)} = \eta, \quad (27)$$

where τ denotes the iteration index. If the functions f_i's are μ-strongly convex, then the convergence rate of SGD is:

$$f(\boldsymbol{w}^{(t+1)}) - f^* \leq \mathcal{O}(\frac{1}{t}) \quad \textit{if} \quad \eta^{(\tau)} = \frac{1}{\mu\tau}, \quad (28)$$

$$f(\boldsymbol{w}^{(t+1)}) - f^* \leq \mathcal{O}\big((1 - \frac{\mu}{L})^t + \eta\big) \quad \textit{if} \quad \eta^{(\tau)} = \eta. \quad (29)$$

Eqs. (27) and (29) show that with a fixed step size η, SGD converges sublinearly for a non-convex function and exponentially for a strongly convex function in the initial iterations. However, in the late iterations where $t \to \infty$, it stagnates to a neighborhood $\mathcal{O}(\eta)$ around the optimal point and never reaches it. For example for Eq. (27), there is:

$$\lim_{t\to\infty} f(\boldsymbol{w}^{(t+1)}) - f^* \leq \lim_{t\to\infty} \mathcal{O}(\frac{1}{t} + \eta) = \mathcal{O}(\eta)$$
$$\implies f(\boldsymbol{w}^{(t+1)}) = f^* + \mathcal{O}(\eta).$$

This is while gradient descent has convergence rate as in Eq. (24) which converges to the solution in the late iterations where $t \to \infty$:

$$\lim_{t\to\infty} f(\boldsymbol{w}^{(t+1)}) - f^* \leq \lim_{t\to\infty} \mathcal{O}(\frac{1}{t}) = \mathcal{O}(0) = 0$$
$$\implies f(\boldsymbol{w}^{(t+1)}) = f^*.$$

Therefore, SGD has less accuracy than gradient descent.

The advantage of SGD over gradient descent is that its every iteration is much faster than every iteration of gradient descent because of less computations for gradient. This faster pacing of every iteration shows off more when n is huge. In summary, SGD has fast convergence to a low accurate optimal solution.

It is noteworthy that the full gradient is not available in SGD to use for checking convergence, as discussed before. One can use other criteria or merely check the norm of gradient for the sampled point. SGD can be used with the line-search methods and momentum, too.

5. Mini-Batch Stochastic Gradient Descent

Gradient descent uses the entire n data points and SGD uses one randomly sampled point at every iteration. For large datasets, gradient descent is very slow and intractable in every iteration while SGD will need a significant number of iterations to roughly cover all data. Besides, SGD has low accuracy in convergence to the optimal solution. There can be a middle case scenario where a batch of b randomly sampled points is used at every iteration. This method is named the mini-batch SGD or the hybrid deterministic-stochastic gradient method. This batch-wise approach is wise for large datasets.

Usually, before start of optimization, the n data points are randomly divided into $\lfloor n/b \rfloor$ batches of size b. This is equivalent to simple random sampling for sampling points into batches without replacement. Suppose the dataset is denoted by by $\mathcal{D}$ (where $|\mathcal{D}| = n$) and the i-th batch is $\mathcal{B}_i$ (where $|\mathcal{B}_i| = b$). The batches are disjoint:

$$\bigcup_{i=1}^{\lfloor n/b \rfloor} \mathcal{B}_i = \mathcal{D}, \quad (30)$$

$$\mathcal{B}_i \cap \mathcal{B}_j = \varnothing, \quad \forall i, j \in \{1, \dots, \lfloor n/b \rfloor\},\ i \neq j. \quad (31)$$

Another less-used approach for making batches is to sample points for a batch during optimization. This is equivalent to bootstrapping for sampling points into batches with replacement. In this case, the batches are not disjoint anymore and Eqs. (30) and (31) do not hold.

Definition 1 (Epoch)**.** *In mini-batch SGD, when all $\lfloor n/b \rfloor$ batches of data are used for optimization once, an epoch is completed. After completion of an epoch, the next epoch is started and epochs are repeated until convergence of optimization.*

In mini-batch SGD, if the k-th iteration of optimization is using the k'-th batch, the update of solution is done as:

$$\boldsymbol{w}^{(k+1)} := \boldsymbol{w}^{(k)} - \eta^{(k)} \frac{1}{b} \sum_{i \in \mathcal{B}_{k'}} \nabla f_i(\boldsymbol{w}^{(k)}). \quad (32)$$

The scale factor $1/b$ is sometimes dropped for simplicity. Mini-batch SGD is used significantly in deep learning and neural networks (Bottou et al., 1998; Goodfellow et al., 2016). Because of dividing data into batches, mini-batch SGD can be solved on parallel servers as a distributed optimization method, making it suitable for optimization in deep learning using GPU cores. Note that the literature and codes of deep learning usually refer to mini-batch SGD as SGD for simplicity and brevity; therefore, do not confuse it with the one-sample SGD discussed in Section 4.

Proposition 3 (Convergence rates for mini-batch SGD)**.** *Consider a function $f(\boldsymbol{w}) = \sum_{i=1}^{n} f_i(\boldsymbol{w})$ which is bounded below and each f_i is differentiable. Let the domain of function $f(.)$ be $\mathcal{D}$ and its gradient be L-smooth and assume $\eta^{(k)} = \eta$ is fixed. The batch-wise gradient is an approximation to the full gradient with some error e_t for the t-th iteration:*

$$\frac{1}{b} \sum_{i \in \mathcal{B}_{t'}} \nabla f_i(\boldsymbol{w}^{(t)}) = \nabla f(\boldsymbol{w}^{(t)}) + e_t. \tag{33}$$

The convergence rate of mini-batch SGD for non-convex and convex functions are:

$$\mathcal{O}(\frac{1}{t} + \|e_t\|_2^2), \tag{34}$$

where t denotes the iteration index. If the functions f_i's are μ-strongly convex, then the convergence rate of mini-batch SGD is:

$$\mathcal{O}\big((1 - \frac{\mu}{L})^t + \|e_t\|_2^2\big). \tag{35}$$

By increasing the batch size, $(1/b) \sum_{i \in \mathcal{B}_{t'}} \nabla f_i(\boldsymbol{w}^{(t)})$ gets closer to $\nabla f(\boldsymbol{w}^{(t)})$ so the error e_t in Eq. (33) is reduced. Therefore, the convergence rate of mini-batch, i.e., Eq. (34), gets closer to that of gradient descent, i.e., Eq. (24), if the batch size increases.

If the batches are sampled without replacement (i.e., sampling batches by simple random sampling before start of optimization) or with replacement (i.e., bootstrapping during optimization), the expected error is (Ghojogh et al., 2020, Proposition 3):

$$\mathbb{E}[\|e_t\|_2^2] = (1 - \frac{b}{n}) \frac{\sigma^2}{b}, \tag{36}$$

$$\mathbb{E}[\|e_t\|_2^2] = \frac{\sigma^2}{b}, \tag{37}$$

respectively, where σ^2 is the variance of whole dataset. According to Eqs. (36) and (37), the accuracy of SGD by sampling without and with replacement increases by $b \to n$ (increasing batch size toward the size of dataset) and $b \to \infty$ (increasing the batch size to infinity), respectively. However, this increase makes every iteration slower so there is a trade-off between accuracy and speed.

6. Adaptive Learning Rate

Recall from Section 2.3 that the suitable learning rate can be found by line-search. However, the learning rate in deep learning is usually set to a initial value and it is adapted in different iterations. The learning rate can be adapted in stochastic gradient descent optimization methods. Three most well-known methods for adapting the learning rate are AdaGrad, RMSProp, and Adam. These adaptive learning rate methods are introduced in the following.

6.1. Adaptive Gradient (AdaGrad)

Adaptive Gradient (AdaGrad) method (Duchi et al., 2011) updates the solution iteratively as:

$$\boldsymbol{w}^{(k+1)} := \boldsymbol{w}^{(k)} - \eta^{(k)} \boldsymbol{G}^{-1} \nabla f_i(\boldsymbol{w}^{(k)}), \tag{38}$$

where $\boldsymbol{G}$ is a $(d \times d)$ diagonal matrix whose (j, j)-th element is:

$$\boldsymbol{G}(j, j) := \sqrt{\varepsilon + \sum_{\tau=0}^{k} \big(\nabla_j f_{i_\tau}(\boldsymbol{w}^{(\tau)})\big)^2}, \tag{39}$$

where $\varepsilon \geq 0$ is for stability (making $\boldsymbol{G}$ full rank), i_τ is the randomly sampled point (from $\{1, \dots, n\}$) at iteration τ, and $\nabla_j f_{i_\tau}(.)$ is the j-th dimension of the derivative of $f_{i_\tau}(.)$; note that $f_{i_\tau}(.)$ is d-dimensional. Putting Eq. (39) in Eq. (38) can simplify AdaGrad to:

$$\boldsymbol{w}_j^{(k+1)} := \boldsymbol{w}_j^{(k)} - \frac{\eta^{(k)}}{\sqrt{\varepsilon + \sum_{\tau=0}^{k} \big(\nabla_j f_{i_\tau}(\boldsymbol{w}^{(\tau)})\big)^2}} \nabla f_j(\boldsymbol{w}_j^{(k)}). \tag{40}$$

AdaGrad keeps a history of the sampled points and it takes derivative for them to use. During the iterations so far, if a dimension has changed significantly, it dampens the learning rate for that dimension (see the inverse in Eq. (38)); hence, it gives more weight for changing the dimensions which have not changed noticeably. In this way, all dimensions will have a fair chance to change.

6.2. Root Mean Square Propagation (RMSProp)

Root Mean Square Propagation (RMSProp) was first proposed by Hinton in (Tieleman & Hinton, 2012) which was an unpublished slide deck for academic lectures in University of Toronto. It is an improved version of Rprop (resilient backpropagation) (Riedmiller & Braun, 1992), which uses the sign of gradient in optimization. Inspired by momentum in Eq. (8):

$$(\Delta \boldsymbol{w})^{(k)} := \alpha (\Delta \boldsymbol{w})^{(k-1)} - \eta^{(k)} \nabla f(\boldsymbol{w}^{(k)}),$$

it updates a scalar variable v as (Hinton et al., 2012):

$$v^{(k+1)} := \gamma v^{(k)} + (1 - \gamma) \|\nabla f_i(\boldsymbol{w}^{(k)})\|_2^2, \tag{41}$$

where $\gamma \in [0, 1]$ is the forgetting factor (e.g., $\gamma = 0.9$). Then, it uses this v to weight the learning rate:

$$\boldsymbol{w}^{(k+1)} := \boldsymbol{w}^{(k)} - \frac{\eta^{(k)}}{\sqrt{\varepsilon + v^{(k+1)}}} \nabla f_j(\boldsymbol{w}_j^{(k)}), \tag{42}$$

where $\epsilon \geq 0$ is for stability not to have division by zero. Comparing Eqs. (40) and (42) shows that RMSProp has a similar form to AdaGrad.

6.3. Adam Optimizer

Adam (Adaptive Moment Estimation) optimizer (Kingma & Ba, 2014) improves over RMSProp by adding a momentum term. It updates the vector $\boldsymbol{m} \in \mathbb{R}^d$ and the scalar v as:

$$\boldsymbol{m}^{(k+1)} := \gamma_1 \boldsymbol{m}^{(k)} + (1 - \gamma_1) \nabla f_i(\boldsymbol{w}^{(k)}), \tag{43}$$

$$v^{(k+1)} := \gamma_2 v^{(k)} + (1 - \gamma_2) \|\nabla f_i(\boldsymbol{w}^{(k)})\|_2^2, \tag{44}$$

where $\gamma_1, \gamma_2 \in [0, 1]$. It normalizes these variables as:

$$\widehat{\boldsymbol{m}}^{(k+1)} := \frac{1}{1 - \gamma_1^k} \boldsymbol{m}^{(k+1)}, \tag{45}$$

$$\widehat{v}^{(k+1)} := \frac{1}{1 - \gamma_2^k} v^{(k+1)}. \tag{46}$$

Then, it updates the solution as:

$$\boldsymbol{w}^{(k+1)} := \boldsymbol{w}^{(k)} - \frac{\eta^{(k)}}{\sqrt{\varepsilon + \widehat{v}^{(k+1)}}} \widehat{\boldsymbol{m}}^{(k+1)}, \tag{47}$$

which is stochastic gradient descent with momentum while using RMSProp. The Adam optimizer is one of the mostly used optimizers in neural networks. In summary, most of the deep learning coding libraries have SGD (i.e., mini-batch SGD) and Adam methods as options of optimizers.

7. Sharpness-Aware Minimization (SAM)

Although backpropagation can find any local minimum of the loss function, not all local minima are equally good. It has been empirically observed (Foret et al., 2021) that the converged local minimum is better to be smooth than sharp; meaning that the neighborhood of the found local minimum is better to be almost flat rather than having a single sharp local minimum (see Fig. 7). It is observed that there may be a connection between smoothness of the found local minimum and generalization of the neural network to unseen test data (Foret et al., 2021). Although, some works have doubted this observation and they state that the smoothness of local minimum is not the only factor for generalization (Wen et al., 2024).

7.1. Vanilla SAM

As a result, Sharpness-Aware Minimization (SAM) has been proposed (Foret et al., 2021). This method uses a zero-sum min-max loss function for finding a flat (smooth) solution while minimizing the loss function of neural network (Foret et al., 2021):

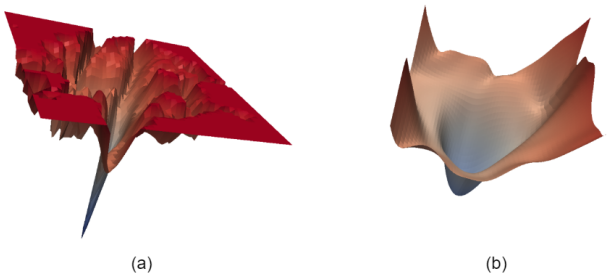

Figure 7. Visual comparison of (a) a sharp local minimum and (b) an almost flat (smooth) local minimum. The credit of image is for (Foret et al., 2021). See (Li et al., 2018) for visualization of loss functions in neural networks.

$$\underset{\boldsymbol{w}}{\text{minimize}}\ \mathcal{L}_{\text{SAM}}(\boldsymbol{w}) + \lambda \|\boldsymbol{w}\|_2^2, \tag{48}$$

$$\mathcal{L}_{\text{SAM}}(\boldsymbol{w}) := \underset{\boldsymbol{\epsilon}: \|\boldsymbol{\epsilon}\|_p \leq \rho}{\text{maximize}}\ \mathcal{L}(\boldsymbol{w} + \boldsymbol{\epsilon}), \tag{49}$$

where $\boldsymbol{w}$ is the weights of neural network, $\mathcal{L}(.)$ is the loss function of neural network, $\mathcal{L}_{\text{SAM}}(.)$ is the SAM loss function, $\lambda \geq 0$ is the regularization parameter, $\rho \geq 0$ is a hyperparameter, and $p \in [1, \infty]$ where $p = 2$ is recommended (Foret et al., 2021). It is possible to drop the regularization term and write the loss as a min-max optimization problem:

$$\underset{\boldsymbol{w}}{\text{minimize}}\ \underset{\boldsymbol{\epsilon}: \|\boldsymbol{\epsilon}\|_p \leq \rho}{\text{maximize}}\ \mathcal{L}(\boldsymbol{w} + \boldsymbol{\epsilon}). \tag{50}$$

The Eq. (50) is first finding the maximum loss in a neighborhood of radius $\boldsymbol{\epsilon}$ around the solution $\boldsymbol{w}$ and then minimizes that. This forces all the local neighborhood of the solution to be small and hence flat or smooth.

Note that Eq. (49) can be restated as (Tahmasebi et al., 2024):

$$\begin{aligned}
\mathcal{L}_{\text{SAM}}(\boldsymbol{w}) &:= \underset{\boldsymbol{\epsilon}: \|\boldsymbol{\epsilon}\|_p \leq \rho}{\text{maximize}}\ \mathcal{L}(\boldsymbol{w} + \boldsymbol{\epsilon}) \\
&= \mathcal{L}(\boldsymbol{w}) - \mathcal{L}(\boldsymbol{w}) + \underset{\boldsymbol{\epsilon}: \|\boldsymbol{\epsilon}\|_p \leq \rho}{\text{maximize}}\ \mathcal{L}(\boldsymbol{w} + \boldsymbol{\epsilon}) \\
&= \underbrace{\mathcal{L}(\boldsymbol{w})}_{\text{empirical loss}} + \underbrace{\underset{\boldsymbol{\epsilon}: \|\boldsymbol{\epsilon}\|_p \leq \rho}{\text{maximize}} \Big(\mathcal{L}(\boldsymbol{w} + \boldsymbol{\epsilon}) - \mathcal{L}(\boldsymbol{w}) \Big)}_{\text{sharpness}}.
\end{aligned} \tag{51}$$

The first term in this equation is the empirical loss and the second term is the sharpness measure. The sharpness term can be generalized into a class of sharpness measurements (Tahmasebi et al., 2024).

Consider the inner maximization in Eq. (50):

$$\begin{aligned}\boldsymbol{\epsilon}^*(\boldsymbol{w}) &:= \arg\max_{\|\boldsymbol{\epsilon}\|_p\leq\rho} \mathcal{L}(\boldsymbol{w}+\boldsymbol{\epsilon}) \\ &\overset{(a)}{\approx} \arg\max_{\|\boldsymbol{\epsilon}\|_p\leq\rho} \mathcal{L}(\boldsymbol{w}) + \boldsymbol{\epsilon}^\top\nabla_{\boldsymbol{w}}\mathcal{L}(\boldsymbol{w}) \\ &\overset{(b)}{=} \arg\max_{\|\boldsymbol{\epsilon}\|_p\leq\rho} \boldsymbol{\epsilon}^\top\nabla_{\boldsymbol{w}}\mathcal{L}(\boldsymbol{w}),\end{aligned}$$

where $\nabla_{\boldsymbol{w}}$ denotes derative with respect to $\boldsymbol{w}$, (a) is because of the first-order Taylor series expansion of $\mathcal{L}(\boldsymbol{w}+\boldsymbol{\epsilon})$ around $\mathbf{0}$ and (b) is because $\mathcal{L}(\boldsymbol{w})$ is not a function of $\boldsymbol{\epsilon}$. This maximization is a classical dual norm problem whose solution is (Foret et al., 2021):

$$\boldsymbol{\epsilon}^*(\boldsymbol{w}) = \rho\,\text{sign}\big(\nabla_{\boldsymbol{w}}\mathcal{L}(\boldsymbol{w})\big)\frac{|\nabla_{\boldsymbol{w}}\mathcal{L}(\boldsymbol{w})|^{q-1}}{(\|\nabla_{\boldsymbol{w}}\mathcal{L}(\boldsymbol{w})\|_q^q)^{(1/p)}}, \quad (52)$$

where $(1/p)+(1/q)=1$. With $p=2$, it simplifies to:

$$\boldsymbol{\epsilon}^*(\boldsymbol{w}) = \rho\,\frac{\nabla_{\boldsymbol{w}}\mathcal{L}(\boldsymbol{w})}{\|\nabla_{\boldsymbol{w}}\mathcal{L}(\boldsymbol{w})\|_2}. \quad (53)$$

It is noteworthy that some discussions exist in the literature that Eq. (53) is an upper bound, and not an exact bound, on the classification error. We refer interested readers to (Xie et al., 2024) for those discussions.

Let us put the found $\boldsymbol{\epsilon}$, i.e., Eq. (53), in Eq. (50) and calculate the gradient of the loss function:

$$\begin{aligned}\nabla_{\boldsymbol{w}}\mathcal{L}(\boldsymbol{w}+\boldsymbol{\epsilon}^*(\boldsymbol{w})) &\overset{(a)}{=} \frac{d(\boldsymbol{w}+\boldsymbol{\epsilon}^*(\boldsymbol{w}))}{d\boldsymbol{w}}\nabla_{\boldsymbol{w}}\mathcal{L}(\boldsymbol{w})\big|_{\boldsymbol{w}+\boldsymbol{\epsilon}^*(\boldsymbol{w})} \\ &= \nabla_{\boldsymbol{w}}\mathcal{L}(\boldsymbol{w})\big|_{\boldsymbol{w}+\boldsymbol{\epsilon}^*(\boldsymbol{w})} + \frac{d\boldsymbol{\epsilon}^*(\boldsymbol{w})}{d\boldsymbol{w}}\nabla_{\boldsymbol{w}}\mathcal{L}(\boldsymbol{w})\big|_{\boldsymbol{w}+\boldsymbol{\epsilon}^*(\boldsymbol{w})} \\ &\overset{(b)}{\approx} \nabla_{\boldsymbol{w}}\mathcal{L}(\boldsymbol{w})\big|_{\boldsymbol{w}+\boldsymbol{\epsilon}^*(\boldsymbol{w})}, \quad (54)\end{aligned}$$

where (a) is because of chain rule in derivatives and (b) is because the second term contains multiplication of two derivatives which is small compared to the first term and hence can be ignored. The Eq. (54) is the gradient of SAM loss function and it can be used in backpropagation. This gradient can be numerically approximated by the deep learning libraries such as PyTorch (Paszke et al., 2019).

7.2. Efficient SAM

Calculation is gradient in the vanilla SAM is time consuming and not efficient. Therefore, Efficient SAM (ESAM) (Du et al., 2022) is proposed for improving the computational efficiency of SAM. ESAM has two methodologies, i.e., Stochastic Weight Perturbation (SWP) and Sharpness-sensitive Data Selection (SDS).

7.2.1. Stochastic Weight Perturbation (SWP)

SWP (Du et al., 2022) efficiently approximates $\boldsymbol{\epsilon}^*(\boldsymbol{w})$ using a random subset of weights rather than all weights of network. Let n denote the number of weights in the neural network and the weights of network be $\{w_1,\dots,w_n\}$. In each iteration of backpropagation, SWP makes a random binary gradient mask $\boldsymbol{m} = [m_1,\dots,m_n]^\top$ where $m_i \overset{\text{i.i.d.}}{\sim} \text{Bernoulli}(\beta)$ where β is the parameter of the Bernoulli distribution. The solution of the inner maximization, i.e., $\boldsymbol{\epsilon}^*(\boldsymbol{w})$, is approximated by (Du et al., 2022):

$$\widehat{\boldsymbol{\epsilon}}(\boldsymbol{w}) \approx \frac{1}{\beta}\boldsymbol{m}^\top\boldsymbol{\epsilon}^*(\boldsymbol{w}), \quad (55)$$

which is used in the gradient of loss in Eq. (54). In other words, SWP does not use the entire weights and, instead, uses a subset of weights for $\boldsymbol{\epsilon}^*(\boldsymbol{w})$. This simplification does not affect the expectation of $\boldsymbol{\epsilon}^*(\boldsymbol{w})$:

$$\begin{aligned}\mathbb{E}[\widehat{\boldsymbol{\epsilon}}(\boldsymbol{w})_i] &\overset{(55)}{=} \frac{1}{\beta}\mathbb{E}[m_i\,\boldsymbol{\epsilon}^*(\boldsymbol{w})_i] \overset{(a)}{=} \frac{1}{\beta}\mathbb{E}[m_i]\,\mathbb{E}[\boldsymbol{\epsilon}^*(\boldsymbol{w})_i] \\ &\overset{(b)}{=} \frac{1}{\beta}\beta\mathbb{E}[\boldsymbol{\epsilon}^*(\boldsymbol{w})_i] = \mathbb{E}[\boldsymbol{\epsilon}^*(\boldsymbol{w})_i],\end{aligned}$$

where $\boldsymbol{\epsilon}(\boldsymbol{w})_i$ denotes the i-th element of $\boldsymbol{\epsilon}(\boldsymbol{w})$, (a) is because m_i and $\boldsymbol{\epsilon}^*(\boldsymbol{w})_i$ are independent, and (b) is because expectation of the Bernoulli distribution of m_i.

7.2.2. Sharpness-sensitive Data Selection (SDS)

SDS (Du et al., 2022) efficiently approximates $\boldsymbol{\epsilon}^*(\boldsymbol{w})$ using a subset of mini-batch rather than the entire mini-batch. It splits the mini-batch $\mathcal{B}$ as (Du et al., 2022):

$$\begin{aligned}\mathcal{B}^+ &:= \{(x_i,y_i)\in\mathcal{B}\,|\,\mathcal{L}(\boldsymbol{w}+\boldsymbol{\epsilon}^*(\boldsymbol{w})) - \mathcal{L}(\boldsymbol{w}) > \alpha\}, \\ \mathcal{B}^- &:= \{(x_i,y_i)\in\mathcal{B}\,|\,\mathcal{L}(\boldsymbol{w}+\boldsymbol{\epsilon}^*(\boldsymbol{w})) - \mathcal{L}(\boldsymbol{w}) < \alpha\},\end{aligned} \quad (56)$$

where $\mathcal{B}^+$ is the sharpness-sensitive subset of mini-batch and $\alpha>0$ is a hyperparameter. SDS uses the sharpness-sensitive $\mathcal{B}^+$ rather than the entire mini-batch $\mathcal{B}$ in the loss function of SAM. Therefore, calculation is faster with less samples in the mini-batch.

7.3. Adaptive SAM

SAM uses a fixed radius when considering $\|\boldsymbol{\epsilon}\|_p \leq \rho$ in its optimization. Therefore, it is sensitive to re-scaling weights of neural network by for example a scaling matrix $\boldsymbol{A}$ (Kwon et al., 2021):

$$\underset{\boldsymbol{\epsilon}:\|\boldsymbol{\epsilon}\|_p\leq\rho}{\text{maximize}}\ \mathcal{L}(\boldsymbol{w}+\boldsymbol{\epsilon}) \neq \underset{\boldsymbol{\epsilon}:\|\boldsymbol{\epsilon}\|_p\leq\rho}{\text{maximize}}\ \mathcal{L}(\boldsymbol{A}\boldsymbol{w}+\boldsymbol{\epsilon}).$$

In other words, neural networks with different weight scales have different sharpness values.

Adaptive SAM (ASAM) (Kwon et al., 2021) makes SAM robust to weight re-scaling. We define a normalization operator of $\boldsymbol{w}$, denoted by $T_{\boldsymbol{w}}^{-1}$, where:

$$T_{\boldsymbol{A}\boldsymbol{w}}^{-1}\boldsymbol{A} = T_{\boldsymbol{w}}^{-1}, \quad (57)$$

for any invertible scaling matrix $\boldsymbol{A}$. ASAM defines and optimizes the adaptive sharpness of $\boldsymbol{w}$ (Kwon et al., 2021):

$$\underset{\boldsymbol{\epsilon}:\|T_{\boldsymbol{w}}^{-1}\boldsymbol{\epsilon}\|_p\leq\rho}{\text{maximize}}\ \mathcal{L}(\boldsymbol{w}+\boldsymbol{\epsilon})-\mathcal{L}(\boldsymbol{w}), \tag{58}$$

while the regular sharpness, sensitive to scale, of $\boldsymbol{w}$ is defined as in Eq. (51).

$$\underset{\boldsymbol{\epsilon}:\|\boldsymbol{\epsilon}\|_p\leq\rho}{\text{maximize}}\ \mathcal{L}(\boldsymbol{w}+\boldsymbol{\epsilon})-\mathcal{L}(\boldsymbol{w}).$$

The adaptive sharpness, defined in Eq. (58), is scale-invariant because for the scaled weights $\boldsymbol{Aw}$, there is:

$$\begin{aligned}
&\underset{\boldsymbol{\epsilon}:\|T_{\boldsymbol{Aw}}^{-1}\boldsymbol{\epsilon}\|_p\leq\rho}{\text{maximize}}\ \mathcal{L}(\boldsymbol{Aw}+\boldsymbol{\epsilon})\overset{(a)}{=}\underset{\boldsymbol{\epsilon}:\|T_{\boldsymbol{Aw}}^{-1}\boldsymbol{\epsilon}\|_p\leq\rho}{\text{maximize}}\ \mathcal{L}(\boldsymbol{w}+\boldsymbol{A}^{-1}\boldsymbol{\epsilon})\\
&\overset{(b)}{=}\underset{\boldsymbol{\epsilon}':\|T_{\boldsymbol{Aw}}^{-1}\boldsymbol{A}\boldsymbol{\epsilon}'\|_p\leq\rho}{\text{maximize}}\ \mathcal{L}(\boldsymbol{w}+\boldsymbol{\epsilon}')\\
&\overset{(57)}{=}\underset{\boldsymbol{\epsilon}':\|T_{\boldsymbol{w}}^{-1}\boldsymbol{\epsilon}'\|_p\leq\rho}{\text{maximize}}\ \mathcal{L}(\boldsymbol{w}+\boldsymbol{\epsilon}')\overset{(c)}{=}\underset{\boldsymbol{\epsilon}:\|T_{\boldsymbol{w}}^{-1}\boldsymbol{\epsilon}\|_p\leq\rho}{\text{maximize}}\ \mathcal{L}(\boldsymbol{w}+\boldsymbol{\epsilon}),
\end{aligned}$$

where (a) is because of left-multiplying the input of loss with $\boldsymbol{A}$, (b) is because of change of variable $\boldsymbol{\epsilon}':=\boldsymbol{A}^{-1}\boldsymbol{\epsilon}$ (so that $\boldsymbol{\epsilon}:=\boldsymbol{A}\boldsymbol{\epsilon}'$), and (c) is because of renaming the dummy variable $\boldsymbol{\epsilon}'$ to $\boldsymbol{\epsilon}$.

8. Convergence Guarantees for Optimization in Over-parameterized Neural Networks

There are some theories on the convergence of optimization in over-parameterized neural networks, i.e., the networks with much more learnable parameters than the number of training data instances. In other words, over-parameterized neural networks are wide networks having sufficient number of neurons in their hidden layers. The assumption of being over-parameterized is for making sure that network has enough capability for learning. These theories explain why shallow (Soltanolkotabi et al., 2018) and deep (Allen-Zhu et al., 2019b;a) neural networks work.

8.1. Convergence Guarantees for Optimization in Shallow Networks

Consider a shallow network with one hidden layer and an output layer. Let d be the dimensionality of input data, k be the number of hidden neurons, and suppose the last layer has one neuron for regressing the label of data. This network maps data $\boldsymbol{x}\in\mathbb{R}^d$ to a scalar output as:

$$\boldsymbol{x}\mapsto\boldsymbol{v}^\top\sigma(\boldsymbol{W}\boldsymbol{x}),$$

where $\boldsymbol{W}\in\mathbb{R}^{k\times d}$ is the weight matrix between input to the hidden layer, $\boldsymbol{v}\in\mathbb{R}^k$ is the weight vector of output layer, and $\sigma(.)$ is the activation function. Suppose the loss function for the data instance $\boldsymbol{x}_i$ is mean squared error between the output of network and the target label y_i:

$$\mathcal{L}(\boldsymbol{W})=\frac{1}{2n}\sum_{i=1}^{n}\big(y_i-\boldsymbol{v}^\top\sigma(\boldsymbol{W}\boldsymbol{x}_i)\big)^2,$$

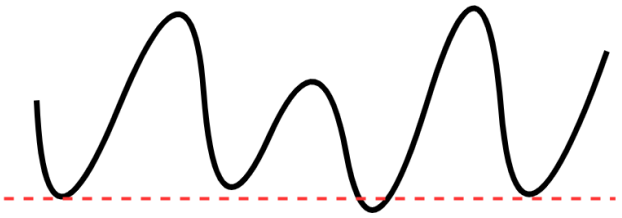

Figure 8. In a neural network, (almost) all local solutions are good enough being the global solution!

where n is the number of training instances.

Proposition 4 ((Soltanolkotabi et al., 2018, Theorem 1))**.** *In the above-mentioned shallow network, suppose the activation function is a quadratic function, i.e., $\sigma(z)=z^2$. If $k\geq 2d$ and the weights of output layer $\boldsymbol{v}$ contains at least d positive values and d negative values, then:*

1. *There are no spurious local minima, i.e., all local minima are global minima.*
2. *All saddle points have a direction of strictly negative curvature. In other words, at a saddle point of weights $\boldsymbol{W}_s$ for the loss function, there is a direction $\boldsymbol{U}\in\mathbb{R}^{k\times d}$ such that:*

$$\boldsymbol{vec}(\boldsymbol{U})^\top\nabla^2\mathcal{L}(\boldsymbol{W}_s)\,\boldsymbol{vec}(\boldsymbol{U})<0,$$

   where $\nabla\mathcal{L}(\boldsymbol{W}_s)$ si the second-order derivative of loss at the saddle point and $\boldsymbol{vec}(.)$ is the vectorization operator which vectorizes the matrix.
3. *If $d\leq n\leq cd^2$, where $c>0$ is a constant, the global optimum of the loss function $\mathcal{L}(\boldsymbol{W})$ is zero.*

There are other forms of Proposition 4, for more relaxed assumptions and general forms of activation function, which can be found in (Soltanolkotabi et al., 2018).

Item 1 in Proposition 4 states that (almost) all local minima of the neural network are good enough, equal to the global minimum. In other words, the situation in Fig. 8 occurs. Therefore, it does not matter much which solution is found based on different initialization of network's weights; the network usually converges to global minimum of the loss function even if it is non-convex (Feizi et al., 2017; Chizat & Bach, 2018; Du et al., 2018; 2019; Chen et al., 2019). Different local solution usually yield to similar performances (Choromanska et al., 2015). This happens because neural network kind of maps data to a reproducing kernel Hilbert space (Ghojogh et al., 2023b) and almost all local minima are global in that higher dimensional space (Dauphin et al., 2014).

Item 2 in Proposition 4 discusses that, in neural networks, there is always a way to escape the saddle points toward local minima by following some directions with negative

curvature. In fact, first-order optimization, including gradient descent used in backpropagation, avoids saddle points for different initializations (Dauphin et al., 2014; Lee et al., 2019; Panageas et al., 2019; Chen et al., 2019).

Item 3 explains that any solution found in the network (which is the global solution according to item 1) is the zero loss for training data. It means that all local solutions can fit the training data perfectly. Although, whether the perfectly working network on training data generalizes well to the unseen test data is another concern, not addressed in this proposition.

8.2. Convergence Guarantees for Optimization in Deep Networks

There also exist convergence guarantees for optimization in deep neural networks (Allen-Zhu et al., 2019b;a).

Proposition 5 ((Allen-Zhu et al., 2019b, Theorems 1 and 2)). *Consider a fully-connected l-layer neural network with mean squared error loss function and ReLU activation function. Without loss of generality*[2]*, assume the data instances are normalized to have unit length and the last dimension of data instances be $1/\sqrt{2}$. Let d be the dimensionality of data and δ be a lower bound on the Euclidean distance of every two points in the training dataset:*

$$\|\boldsymbol{x}_i - \boldsymbol{x}_j\|_2 \geq \delta, \quad \forall i, j \in \{1, \dots, n\},$$

where n is the number of training data instances. Assume the weights of network are randomly initialized.

- *Consider a parameter $m \geq \Omega(\textbf{poly}(n, l, \delta^{-1})\, d)$, where $\Omega(.)$ denotes the lower bound complexity and $\textbf{poly}(.)$ is a polynomial function of its input arguments. Having the learning rate $\eta = \Theta(\frac{d\delta}{\textbf{poly}(n,l)\, m})$, gradient descent converges to a small loss value less than ϵ after:*

$$T = \Theta\Big(\frac{\textbf{poly}(n, l) \log(\epsilon^{-1})}{\delta^2}\Big), \tag{59}$$

  iterations, with high probability at least $1 - e^{-\Omega(\log^2(m))}$.

- *Consider a parameter $m \geq \Omega(\frac{\textbf{poly}(n,l,\delta^{-1})\, d}{b})$, where $b \in \{1, \dots, n\}$ is the mini-batch size. Having the learning rate $\eta = \Theta(\frac{bd\delta}{\textbf{poly}(n,l)\, m \log^2(m)})$, mini-batch SGD converges to a small loss value less than ϵ after:*

$$T = \Theta\Big(\frac{\textbf{poly}(n, l) \log(\epsilon^{-1}) \log^2(m)}{\delta^2 b}\Big), \tag{60}$$

  iterations, with high probability at least $1 - e^{-\Omega(\log^2(m))}$.

[2] It is always possible to normalize data and also add an auxiliary dimension with value $1/\sqrt{2}$.

Proposition 5 explains that the deep over-parameterized neural networks converge to the solution, with a sufficiently small loss value, in polynomial time.

8.3. Other Works on Convergence Guarantees for Optimization in Neural Networks

Note that convergence guarantees for optimization in neural networks using different activation functions have been discussed in the literature. For example, in addition to the above-mentioned references, convergence guarantees for networks with quadratic activation (Du & Lee, 2018), ReLU activation (Cao & Gu, 2020; Zhang et al., 2019; Sharifnassab et al., 2020), and leaky ReLU (Brutzkus et al., 2017) exist.

Convergence analysis have also been done for other network structures, such as ResNet (Du et al., 2018; Shamir, 2018), and other types of data such as structured data (Li & Liang, 2018). Analysis of critical points, i.e., points in which the sign of gradient of loss function changes, has also been discussed for neural networks (Zhou & Liang, 2017; Nouiehed & Razaviyayn, 2022). Moreover, convergence analysis for binary classification (Liang et al., 2018), loss surface analysis with an algebraic geometry approach (Mehta et al., 2021), error bounds on gradient descent in networks (Cao & Gu, 2019) exist in the literature.

Acknowledgement

Some of the materials in this tutorial paper have been covered by Prof. Ali Ghodsi's (Data Science Courses) and Benyamin Ghojogh's videos on YouTube. Moreover, some parts of this tutorial paper were inspired by the lectures of Prof. Kimon Fountoulakis at the University of Waterloo.

References

Allen-Zhu, Zeyuan, Li, Yuanzhi, and Liang, Yingyu. Learning and generalization in overparameterized neural networks, going beyond two layers. *Advances in neural information processing systems*, 32, 2019a.

Allen-Zhu, Zeyuan, Li, Yuanzhi, and Song, Zhao. A convergence theory for deep learning via over-parameterization. In *International conference on machine learning*, pp. 242–252. PMLR, 2019b.

Armijo, Larry. Minimization of functions having Lipschitz continuous first partial derivatives. *Pacific Journal of mathematics*, 16(1):1–3, 1966.

Bottou, Léon et al. Online learning and stochastic approximations. *On-line learning in neural networks*, 17(9): 142, 1998.

Brutzkus, Alon, Globerson, Amir, Malach, Eran, and Shalev-Shwartz, Shai. SGD learns over-parameterized

networks that provably generalize on linearly separable data. *arXiv preprint arXiv:1710.10174*, 2017.

Cao, Yuan and Gu, Quanquan. Generalization bounds of stochastic gradient descent for wide and deep neural networks. *Advances in neural information processing systems*, 32, 2019.

Cao, Yuan and Gu, Quanquan. Generalization error bounds of gradient descent for learning over-parameterized deep relu networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pp. 3349–3356, 2020.

Chen, Yuxin, Chi, Yuejie, Fan, Jianqing, and Ma, Cong. Gradient descent with random initialization: Fast global convergence for nonconvex phase retrieval. *Mathematical Programming*, 176:5–37, 2019.

Chizat, Lenaic and Bach, Francis. On the global convergence of gradient descent for over-parameterized models using optimal transport. *Advances in neural information processing systems*, 31, 2018.

Chong, Edwin KP and Zak, Stanislaw H. *An introduction to optimization*. John Wiley & Sons, 2004.

Choromanska, Anna, Henaff, Mikael, Mathieu, Michael, Arous, Gérard Ben, and LeCun, Yann. The loss surfaces of multilayer networks. In *Artificial intelligence and statistics*, pp. 192–204. PMLR, 2015.

Curry, Haskell B. The method of steepest descent for non-linear minimization problems. *Quarterly of Applied Mathematics*, 2(3):258–261, 1944.

Dauphin, Yann N, Pascanu, Razvan, Gulcehre, Caglar, Cho, Kyunghyun, Ganguli, Surya, and Bengio, Yoshua. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. *Advances in neural information processing systems*, 27, 2014.

Du, Jiawei, Yan, Hanshu, Feng, Jiashi, Zhou, Joey Tianyi, Zhen, Liangli, Goh, Rick Siow Mong, and Tan, Vincent YF. Efficient sharpness-aware minimization for improved training of neural networks. In *International Conference on Learning Representations (ICLR)*, 2022.

Du, Simon and Lee, Jason. On the power of over-parametrization in neural networks with quadratic activation. In *International conference on machine learning*, pp. 1329–1338. PMLR, 2018.

Du, Simon, Lee, Jason, Li, Haochuan, Wang, Liwei, and Zhai, Xiyu. Gradient descent finds global minima of deep neural networks. In *International conference on machine learning*, pp. 1675–1685. PMLR, 2019.

Du, Simon S, Zhai, Xiyu, Poczos, Barnabas, and Singh, Aarti. Gradient descent provably optimizes over-parameterized neural networks. *arXiv preprint arXiv:1810.02054*, 2018.

Duchi, John, Hazan, Elad, and Singer, Yoram. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12 (7), 2011.

Feizi, Soheil, Javadi, Hamid, Zhang, Jesse, and Tse, David. Porcupine neural networks:(almost) all local optima are global. *arXiv preprint arXiv:1710.02196*, 2017.

Foret, Pierre, Kleiner, Ariel, Mobahi, Hossein, and Neyshabur, Behnam. Sharpness-aware minimization for efficiently improving generalization. In *International Conference on Learning Representations (ICLR)*, 2021.

Ghojogh, Benyamin, Nekoei, Hadi, Ghojogh, Aydin, Karray, Fakhri, and Crowley, Mark. Sampling algorithms, from survey sampling to Monte Carlo methods: Tutorial and literature review. *arXiv preprint arXiv:2011.00901*, 2020.

Ghojogh, Benyamin, Ghodsi, Ali, Karray, Fakhri, and Crowley, Mark. KKT conditions, first-order and second-order optimization, and distributed optimization: Tutorial and survey. *arXiv preprint arXiv:2110.01858*, 2021.

Ghojogh, Benyamin, Crowley, Mark, Karray, Fakhri, and Ghodsi, Ali. Background on optimization. In *Elements of Dimensionality Reduction and Manifold Learning*, pp. 75–120. Springer, 2023a.

Ghojogh, Benyamin, Crowley, Mark, Karray, Fakhri, and Ghodsi, Ali. Background on kernels. *Elements of Dimensionality Reduction and Manifold Learning*, pp. 43–73, 2023b.

Goodfellow, Ian, Bengio, Yoshua, and Courville, Aaron. *Deep learning*. MIT press, 2016.

Hadamard, Jacques. *Mémoire sur le problème d'analyse relatif à l'équilibre des plaques élastiques encastrées*, volume 33. Imprimerie nationale, 1908.

Hinton, Geoffrey, Srivastava, Nitish, and Swersky, Kevin. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. Technical report, Department of Computer Science, University of Toronto, 2012.

Hinton, Geoffrey E and Salakhutdinov, Ruslan R. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

Kingma, Diederik P and Ba, Jimmy. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Kwon, Jungmin, Kim, Jeongseop, Park, Hyunseo, and Choi, In Kwon. ASAM: Adaptive sharpness-aware minimization for scale-invariant learning of deep neural networks. In *International Conference on Machine Learning*, pp. 5905–5914. PMLR, 2021.

Lee, Jason D, Panageas, Ioannis, Piliouras, Georgios, Simchowitz, Max, Jordan, Michael I, and Recht, Benjamin. First-order methods almost always avoid strict saddle points. *Mathematical programming*, 176:311–337, 2019.

Lemaréchal, Claude. Cauchy and the gradient method. *Doc Math Extra*, 251(254):10, 2012.

Leung, Frank Hung-Fat, Lam, Hak-Keung, Ling, Sai-Ho, and Tam, Peter Kwong-Shun. Tuning of the structure and parameters of a neural network using an improved genetic algorithm. *IEEE Transactions on Neural networks*, 14(1):79–88, 2003.

Li, Hao, Xu, Zheng, Taylor, Gavin, Studer, Christoph, and Goldstein, Tom. Visualizing the loss landscape of neural nets. *Advances in neural information processing systems*, 31, 2018.

Li, Yuanzhi and Liang, Yingyu. Learning overparameterized neural networks via stochastic gradient descent on structured data. *Advances in neural information processing systems*, 31, 2018.

Liang, Shiyu, Sun, Ruoyu, Li, Yixuan, and Srikant, Rayadurgam. Understanding the loss surface of neural networks for binary classification. In *International Conference on Machine Learning*, pp. 2835–2843. PMLR, 2018.

Mehta, Dhagash, Chen, Tianran, Tang, Tingting, and Hauenstein, Jonathan D. The loss surface of deep linear networks viewed through the algebraic geometry lens. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(9):5664–5680, 2021.

Montana, David J and Davis, Lawrence. Training feedforward neural networks using genetic algorithms. In *IJCAI*, volume 89, pp. 762–767, 1989.

Nouiehed, Maher and Razaviyayn, Meisam. Learning deep models: Critical points and local openness. *INFORMS Journal on Optimization*, 4(2):148–173, 2022.

Panageas, Ioannis, Piliouras, Georgios, and Wang, Xiao. First-order methods almost always avoid saddle points: The case of vanishing step-sizes. *Advances in Neural Information Processing Systems*, 32, 2019.

Paszke, Adam, Gross, Sam, Massa, Francisco, Lerer, Adam, Bradbury, James, Chanan, Gregory, Killeen, Trevor, Lin, Zeming, Gimelshein, Natalia, Antiga, Luca, et al. PyTorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Riedmiller, Martin and Braun, Heinrich. Rprop-a fast adaptive learning algorithm. In *Proceedings of the International Symposium on Computer and Information Science VII*, 1992.

Robbins, Herbert and Monro, Sutton. A stochastic approximation method. *The annals of mathematical statistics*, pp. 400–407, 1951.

Rumelhart, David E, Hinton, Geoffrey E, and Williams, Ronald J. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

Shamir, Ohad. Are ResNets provably better than linear predictors? *Advances in neural information processing systems*, 31, 2018.

Sharifnassab, Arsalan, Salehkaleybar, Saber, and Golestani, S Jamaloddin. Bounds on over-parameterization for guaranteed existence of descent paths in shallow ReLU networks. In *International conference on learning representations*, 2020.

Soltanolkotabi, Mahdi, Javanmard, Adel, and Lee, Jason D. Theoretical insights into the optimization landscape of over-parameterized shallow neural networks. *IEEE Transactions on Information Theory*, 65(2):742–769, 2018.

Tahmasebi, Behrooz, Soleymani, Ashkan, Bahri, Dara, Jegelka, Stefanie, and Jaillet, Patrick. A universal class of sharpness-aware minimization algorithms. In *International Conference on Machine Learning (ICML)*, 2024.

Tieleman, Tijmen and Hinton, Geoffrey. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

Wen, Kaiyue, Li, Zhiyuan, and Ma, Tengyu. Sharpness minimization algorithms do not only minimize sharpness to achieve better generalization. *Advances in Neural Information Processing Systems*, 36, 2024.

Wolfe, Philip. Convergence conditions for ascent methods. *SIAM review*, 11(2):226–235, 1969.

Xie, Wanyun, Latorre, Fabian, Antonakopoulos, Kimon, Pethick, Thomas, and Cevher, Volkan. Improving SAM requires rethinking its optimization formulation. In *International Conference on Machine Learning (ICML)*, 2024.

Zhang, Xiao, Yu, Yaodong, Wang, Lingxiao, and Gu, Quanquan. Learning one-hidden-layer ReLU networks via gradient descent. In *The 22nd international conference on artificial intelligence and statistics*, pp. 1524–1534. PMLR, 2019.

Zhou, Yi and Liang, Yingbin. Critical points of neural networks: Analytical forms and landscape properties. *arXiv preprint arXiv:1710.11205*, 2017.