1 Gradient-Based Learning Algorithms

1.1 Introduction

Once you have specified a learning problem (loss function, hypothesis space, parameterization), the next step is to find the parameters that minimize the loss. This is an optimization problem, and the most common optimization algorithm we will use is **gradient descent**. Gradient descent is like a skier making their way down a snowy mountain, where the shape of the mountain is the loss function.

There are many varieties of gradient descent, and we will call this whole family **gradient-based learning algorithms**. All share the same basic idea: at some operating point, calculate the direction of steepest descent, then use this direction to find a new operating point with lower loss.

We use the term **operating point** to refer to a particular point (setting of the parameters) where we are currently evaluating the loss.

1.2 Technical Setting

In this chapter, we consider the task of minimizing a cost function $J : \cdot \to \mathbb{R}$, which is a function that maps some arbitrary input to a scalar cost.

In learning problems, the domain of J is the training data and the parameters θ. Often, we will consider the training data to be fixed and only denote the objective as a function of the parameters, $J(\theta)$. Our goal is to solve:

$$\theta^* = \arg\min_{\theta} J(\theta) \tag{1.1}$$

Remember from chapter **??** that in supervised learning, the training data is $\{\mathbf{x}^{(i)}, \mathbf{y}^{(i)}\}_{i=1}^{N}$, while in unsupervised learning the training data is $\{\mathbf{x}^{(i)}\}_{i=1}^{N}$.

Pretty much all optimizers work by some iterative process, where they update the parameters to be better and better. Different optimizers differ in how the parameter update function works. The update function gets to view some information about the loss landscape, then uses that information to update the parameters, as shown in figure 1.1.

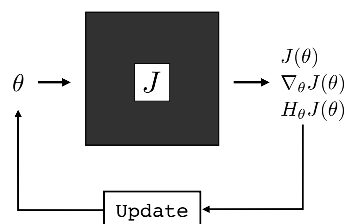

Figure 1.1: General optimization loop.

In the simplest setting, called **zeroth-order optimization**, the update function only gets to observe the value $J(\theta)$. The only way, then, to find θ's that minimize the loss is to sample different values for θ and move toward the values that are lower.

For gradient-based optimization, also called **first-order optimization**, the update function takes as input the gradient of the parameters at the current operating point, $\nabla_\theta J(\theta)$. This reveals hugely useful information about the loss that directly tells you how to minimize it: just move in the direction of steepest descent, that is, the gradient direction.

Higher-order optimization methods observe higher-order derivatives of the loss, such as the Hessian H, which tells you how the landscape is locally curving. The Hessian is costly to compute but many methods use approximations to the Hessian, or other properties related to loss curvature, and these are growing in popularity [6, 2].

1.3 Basic Gradient Descent

The simplest version of gradient descent just takes a step in the gradient direction of length proportional to the gradient magnitude. This algorithm is described in algorithm 1.1.

Algorithm 1.1: Optimizing a cost function $J : \theta \rightarrow \mathbb{R}$ by descending the gradient $\nabla_\theta J$.

Algorithm 1.1: Gradient descent (`GD`)

1. **Input:** objective function J, initial parameter vector θ^0, learning rate η, number of steps K
2. **Output:** trained parameter vector $\theta^* = \theta^K$
3. **for** $k = 0, \ldots, K-1$ **do**
4. $\quad \theta^{k+1} \leftarrow \theta^k - \eta \nabla_\theta J(\theta^k)$

This algorithm has two hyperparameters, the **learning rate** η, which controls the step size (learning rate times gradient magnitude), and the number of steps K. If the learning rate is sufficiently small and the initial parameter vector θ^0 is random, then this algorithm will almost surely converge to a local minimum of J as $K \rightarrow \infty$ [4]. However, to descend more quickly, it can be useful to set the learning rate to a higher value.

1.4 Learning Rate Schedules

A generally useful strategy is to start with a high value for η and then decay it until convergence according to a **learning rate schedule**. Researchers have come up with innumerable schedules and they generally work by calling some function $\texttt{lr}(\eta^0, k)$ to get the learning rate on each iteration of descent:

$$\eta^k = \texttt{lr}(\eta^0, k) \tag{1.2}$$

Generally, we want an update rule where $\eta^{k+1} < \eta^k$, so that we take smaller steps as we approach the minimizer. A few simple and popular approaches are given below:

$$\texttt{lr}(\eta^0, k) = \beta^{-k}\eta^0 \quad \triangleleft \text{ exponential decay} \tag{1.3}$$

$$\texttt{lr}(\eta^0, k) = \beta^{-\lfloor k/M \rfloor}\eta^0 \quad \triangleleft \text{ stepwise exponential decay} \tag{1.4}$$

$$\texttt{lr}(\eta^0, k) = \frac{(K-k)}{K}\eta^0 \quad \triangleleft \text{ linear decay} \tag{1.5}$$

The β and M are additional hyperparameters of these methods. The general approach to learning rate decay is summarized in algorithm 1.2.

One downside of linear decay is that it depends on the total number of steps K. This makes it hard to compare optimization runs of different lengths. This is something to also be aware of in more advanced learning rate schedules, such as cosine decay [5], which also have different behavior for different settings of K.

Algorithm 1.2: `GD+lr`-schedule.

1 **Input:** objective function J, initial parameter vector θ^0, initial learning rate η^0, learning rate function `lr`, number of steps K
2 **Output:** trained parameter vector $\theta^* = \theta^K$
3 **for** $k = 0, \ldots, K-1$ **do**
4 $\quad \eta^k \leftarrow \texttt{lr}(\eta^0, k)$
5 $\quad \theta^{k+1} \leftarrow \theta^k - \eta^k \nabla_\theta J(\theta^k)$

Algorithm 1.2: Gradient descent with a learning rate schedule.

Variations on this algorithm include only decaying the learning rate when a plateau is reached (i.e., when the loss is not decreasing for many iterations in a row), or decaying the learning rate according to more complex nonlinear schedules, such as one shaped like a cosine function [5].

1.5 Momentum

Could we do a smarter update than just taking a step in the direction of the gradient? Of the countless ideas that have been proposed, one of the few that has stuck is **momentum** [9, 11]. Momentum makes the analogy to skiing even more precise: momentum is like the inertia of the skier, carrying them over the little bumps and imperfections in the ski slope and increasing their speed as they descend along a straight path. In math, momentum just means that we set the parameter update to be a direction $\mathbf{v}^{k+1}$, given by a weighted combination of the previous update direction, $\mathbf{v}^k$, plus the current negative gradient:

$$\mathbf{v}^{k+1} = \mu\mathbf{v}^k - \eta\nabla_\theta J(\theta^k) \tag{1.6}$$

The weight μ in this combination is a new hyperparameter, sometimes simply called the "momentum." The full algorithm is given in algorithm 1.3.

Figure 1.2 shows how momentum affects gradient descent for a simple objective $J =$ `abs`(θ). As can be seen in the figure, some momentum can help convergence rate (figure 1.2, $\mu = 0.5$) but too much momentum will cause the trajectory to overshoot the optimum and even when the optimum loss is achieved, the trajectory might not stop (figure 1.2, $\mu = 0.95$).

Algorithm 1.3: GD+momentum.

1 **Input:** objective function J, initial parameter vector θ^0, learning rate η, momentum μ, number of steps K
2 **Output:** trained parameter vector $\theta^* = \theta^K$
3 $\mathbf{v}^0 = \mathbf{0}$
4 **for** $k = 0, \ldots, K-1$ **do**
5 $\quad \mathbf{v}^{k+1} = \mu \mathbf{v}^k - \eta \nabla_\theta J(\theta^k)$
6 $\quad \theta^{k+1} \leftarrow \theta^k + \mathbf{v}^{k+1}$

Algorithm 1.3: Gradient descent with momentum.

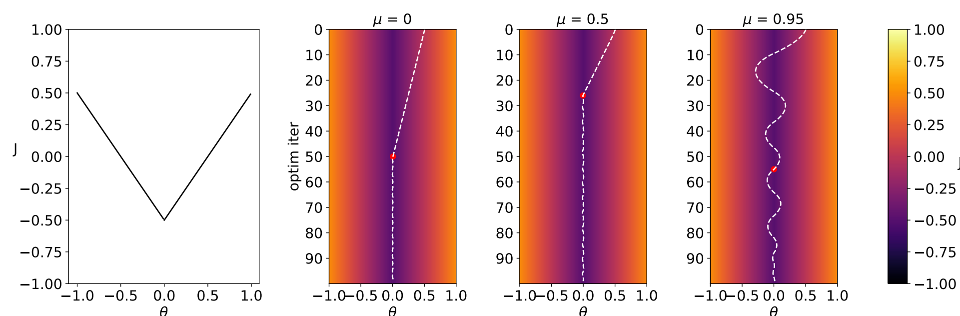

Figure 1.2: (left) A simple loss function $J = \texttt{abs}(\theta)$. (right) Optimization trajectory for three different settings of momentum μ. White line indicates value of the parameter at each iteration of optimization, starting at top and progressing to bottom. Color is value of the loss. Red dot is location where loss first reaches within 0.01 of optimal value.

It is also possible to come up with other kinds of momentum, which bias the update direction based on some accumulated information from previous updates. Two popular alternatives, which you can read up on elsewhere, are Nesterov's accelerated gradient [7] and Adam [3].

1.6 What Kinds of Functions Can Be Minimized with Gradient Descent?

What if a function is not differentiable? Can we still use gradient descent? Sometimes! The property we need is that we can get a meaningful signal as to how to perturb the function's parameters in order to reduce the loss. This property is *not* the same as differentiability defined in math textbooks. A function may be differentiable but not give useful gradients (e.g., if the gradient is zero everywhere), and a function may be nondifferentiable (at certain points) but still allow for meaningful gradient-based updates (e.g., a `relu`).

Figure 1.3 gives examples of different types of functions being minimized with gradient descent. Figures 1.3(b) and 1.3(c) are cases where the function is discontinuous, and the analytical derivative is undefined at the discontinuity. Surprisingly, in figure 1.3(b), this is not a problem for gradient descent. This is because the gradient descent algorithm we are using here (the one used by Pytorch [8]) uses a **one-sided derivative** at the discontinuity, that is, the gradient an infinitesimal step away from the discontinuouity in a fixed arbitrary direction. Under the hood, for each atomic discontinuous operation, Pytorch requires that

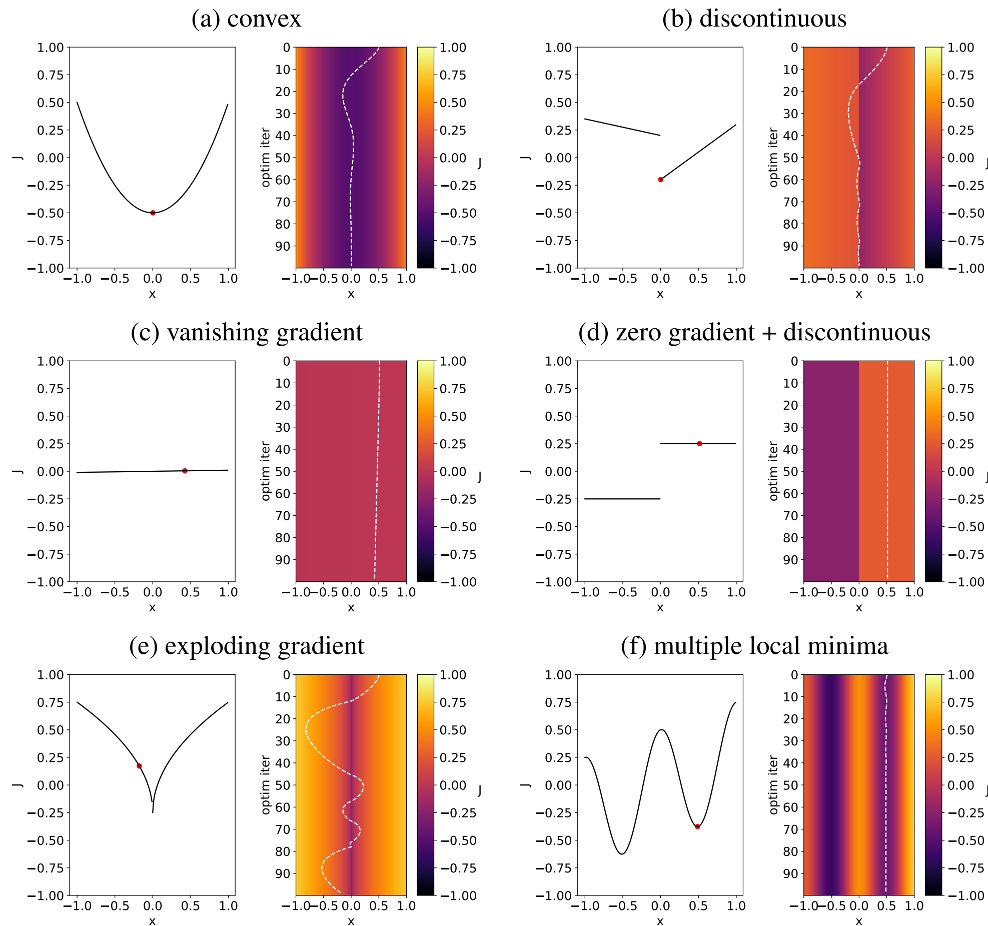

Figure 1.3
How gradient descent behaves on various functions. In each subplot, left is the function J, the red point is the solution found by GD (with $\eta = 0.01$ and $\mu = 0.9$), and right is the trajectory of x values over iterations of GD, plotted on top of J at each iteration. (a) As η goes to zero, GD converges for convex functions. (b) Discontinuities pose no essential problem, as long as the gradient is defined on either side of the discontinuity. (c) A nearly flat function will exhibit very slow descent; we say the gradient has nearly **vanished**. (d) Piecewise constant functions are problematic because the gradient completely vanishes; it is zero everywhere except at the discontinuity, where it is undefined. (e) For the function $y = \texttt{sqrt}(\texttt{abs}(x)) - 0.25$, the gradient goes to infinity at the minimizer; we call this an **exploding gradient**, which leads to failures of convergence. (f) When y has multiple local minima, where we initialize x will determine which minimum we find; we may not find the global minimum.

we define its gradients at the discontinuouities, and the one-sided gradient is a standard choice. This is why it can be fine in deep learning to use functions like the ubiquitous `relu`, which have discontinuous gradients.

1.6.1 Gradient-Like Optimization for Functions without Good Gradients

What about minimizing functions like figure 1.3(d), where the gradient is zero almost everywhere? This is a case where gradient descent truly struggles. However, it is often possible to transform such a problem into one that can be treated with gradient descent. Remember that the key property of a gradient, from the perspective of optimization, is that it is a locally loss-minimizing direction in parameter space. Most gradient-based optimizers don't really need true gradients; instead their update functions are compatible with a broader family of local loss-minimizing directions, $\mathbf{v}$.

Besides the true gradient, what are some other good choices for $\mathbf{v}$? One common idea is to set $\mathbf{v}$ to be the gradient of a **surrogate loss** function (which is a function, $J_{\texttt{surr}}$, with meaningful [non-zero] gradients) that approximates J. An example might be a smoothed version of J. Another way to get $\mathbf{v}$ is to compute it by sampling perturbations of θ, and seeing which perturbation leads to lower loss. In this strategy, we evaluate $J(\theta+\epsilon)$ for a set of perturbations ϵ, then move toward the ϵ's that decreased the loss. Approaches of this kind are sometimes called **evolution strategies** [1, 10], and a basic version of this algorithm is given in algorithm 1.4:

Algorithm 1.4: Evolution strategies (ES).

1 **Input:** objective function J, initial parameter vector θ^0, learning rate η, sampling SD σ, number of samples M, number of steps K
2 **Output:** trained parameter vector $\theta^* = \theta^K$
3 **for** $k = 0, \ldots, K-1$ **do**
4   **for** $i = 1, \ldots, M$ **do**
5     $\epsilon_i \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
6     $s_i = J(\theta + \sigma\epsilon_i)$
7   $\theta^{k+1} \leftarrow \theta^k - \eta \frac{1}{\sigma M} \sum_{i=1}^{M} s_i \epsilon_i$

Algorithm 1.4: Optimizing a cost function $J : \theta \to \mathbb{R}$ by evolution strategies, i.e., sampling different values for θ and taking a step toward the values that work best.

As shown in figure 1.4, this algorithm can successfully minimize the function in figure 1.3(c).

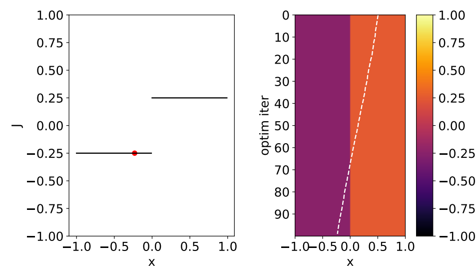

Figure 1.4: Using ES (algorithm 1.4) to minimize a nondifferentiable (zero-gradient) loss, using $\sigma = 1$, $M = 10$, and $\eta = 0.02$.

1.6.2 Gradient Clipping

What about figure 1.3(d), where the gradient explodes near the optimum? Is there anything we can do to improve optimization of this function? To combat exploding gradients, a useful trick is **gradient clipping**, which just means clamping the magnitude of the gradient to some maximum value. Algorithm 5 describes this approach.

Algorithm 1.5: `GD+clip`.

1 **Input:** objective function J, initial parameter vector θ^0, learning rate η, number of steps K, max gradient magnitude m
2 **Output:** trained parameter vector $\theta^* = \theta^K$
3 **for** $k = 0, \ldots, K-1$ **do**
4 $\quad \mathbf{v} = \nabla_\theta J(\theta^k)$
5 $\quad \theta^{k+1} \leftarrow \theta^k - \eta[\texttt{clip}(v_1, -m, m), \ldots, \texttt{clip}(v_M, -m, m)]^\mathsf{T}$

Algorithm 1.5: Gradient descent with gradient clipping.

`clip` is the "clipping" function: $\texttt{clip}(v, -m, m) = \max(\min(v, m), -m)$

This algorithm indeed successfully minimizes our example of the exploding gradient, as can be seen in figure 1.5.

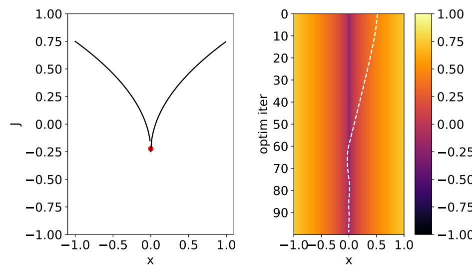

Figure 1.5: Using `GD` with clipping to minimize a loss with exploding gradients, using $m = 0.1$.

1.7 Stochastic Gradient Descent

One problem with the gradient-based methods we have seen so far is that the gradient may in fact be very expensive to compute, and this is often the case for learning problems. This is because learning problems typically have the form that J is the average of losses incurred on each training datapoint. Computing $\nabla_\theta J(\theta)$ requires computing the gradient for each element in the average, that is, the gradient of the function being learned evaluated at the location of each datapoint in the training set. If we train on a big dataset, say 1 million training points, then to perform just *one* step of gradient descent requires computing 1 million gradients! To make this clear, we will write out J as an explicit function of the training data $\{\mathbf{x}^{(i)}, \mathbf{y}^{(i)}\}_{i=1}^N$. For typical learning problems, $\nabla_\theta J(\theta, \{\mathbf{x}^{(i)}, \mathbf{y}^{(i)}\}_{i=1}^N)$ decomposes

as follows:

$$\nabla_\theta J(\theta, \{\mathbf{x}^{(i)}, \mathbf{y}^{(i)}\}_{i=1}^N) = \nabla_\theta \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(f_\theta(\mathbf{x}^{(i)}), \mathbf{y}^{(i)}) \tag{1.7}$$

$$= \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta \mathcal{L}(f_\theta(\mathbf{x}^{(i)}), \mathbf{y}^{(i)}) \tag{1.8}$$

For large N, computing this sum is very expensive. Suppose instead we randomly subsample (without replacement) a *batch* of terms from this sum, $\{\mathbf{x}^{(b)}, \mathbf{y}^{(b)}\}_{b=1}^B$. We then compute an *estimate* of the total gradient as the average gradient over this batch as:

$$\tilde{\mathbf{g}} = \frac{1}{N} \sum_{b=1}^{B} \nabla_\theta \mathcal{L}(f_\theta(\mathbf{x}^{(b)}), \mathbf{y}^{(b)}) \tag{1.9}$$

If we sample a large batch, where B is almost as large as N, then the average over the B terms should be roughly the same as the average over all N terms. If we sample a smaller batch, then our estimate of the gradient will be less accurate but faster to compute. Therefore we have a tradeoff between accuracy and speed, and we can navigate this tradeoff with the hyperparameter B. The variant of gradient descent that uses this idea is called **stochastic gradient descent** (SGD), because each iteration of descent uses a different randomly (stochastically) sampled batch of training data to estimate the gradient. The full description of `SGD` is given in algorithm 1.6.

Algorithm 1.6: Stochastic gradient descent estimates the gradient from a stochastic subset (batch) of the full training data, and makes an update on that basis.

Algorithm 1.6: Stochastic gradient descent (`SGD`).

1. **Input:** initial parameter vector θ^0, data $\{\mathbf{x}^{(i)}, \mathbf{y}^{(i)}\}_{i=1}^N$, learning rate η, batch size B, number of steps K
2. **Output:** trained parameter vector $\theta^* = \theta^K$
3. **for** $k = 0, \ldots, K-1$ **do**
4. $\quad \{\mathbf{x}^{(b)}, \mathbf{y}^{(b)}\}_{b=1}^B \sim \{\mathbf{x}^{(i)}, \mathbf{y}^{(i)}\}_{i=1}^N$ ◁ sample batch of training data
5. $\quad \tilde{\mathbf{g}} = \frac{1}{N} \sum_{b=1}^{B} \nabla_\theta \mathcal{L}(f_\theta(\mathbf{x}^{(b)}), \mathbf{y}^{(b)})$
6. $\quad \theta^k \leftarrow \theta^{k-1} - \eta \tilde{\mathbf{g}}$

`SGD` has a number of useful properties beyond just being faster to compute than `GD`. Because each step of descent is somewhat random, `SGD` can jump over small bumps in the loss landscape, as long those bumps disappear for some randomly sampled batches. Another important property is that `SGD` can implicitly regularize the learning problem. For example, for linear problems (i.e., f_θ is linear), then if there are multiple parameter settings that minimize the loss, `SGD` will often converge to the solution with minimum parameter norm [12].

1.8 Concluding Remarks

The study of optimization can fill dozens of textbooks and thousands of academic papers. But fortunately for us, modern machine learning has converged on just a few very simple optimization methods that are used in practice. In deep learning, in particular, gradient-based optimization is the workhorse. Sometimes this is even treated as definitional to deep learning: deep learning is learning with gradient-based optimization. Believe it or not, the handful of algorithms described above are enough to train most state-of-the-art deep learning models. Every year there are new elaborations on these ideas, and second-order methods are ever on the horizon, yet the basic concepts remain quite simple: compute a local estimate of the shape of the loss landscape, then, based on this shape, take a small step toward a lower loss.

References

[1] Hans-Georg Beyer and Hans-Paul Schwefel. "Evolution Strategies–a Comprehensive Introduction". In: *Natural Computing* 1.1 (2002), pp. 3–52.

[2] Pierre Foret et al. "Sharpness-Aware Minimization for Efficiently Improving Generalization". In: *ICLR* (2021).

[3] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: (2014). https://arxiv.org/abs/1412.6.

[4] Jason D Lee et al. "Gradient Descent Only Converges to Minimizers". In: *Conference on Learning Theory*. PMLR. 2016, pp. 1246–1257.

[5] Ilya Loshchilov and Frank Hutter. "Sgdr: Stochastic Gradient Descent with Warm Restarts". In: *ICLR* (2017).

[6] James Martens and Roger Grosse. "Optimizing Neural Networks with Kronecker-Factored Approximate Curvature". In: *International Conference on Machine Learning*. PMLR. 2015, pp. 2408–2417.

[7] Yurii Nesterov. "A Method for Unconstrained Convex Minimization Problem with the Rate of Convergence O $(1/k^2)$". In: *Doklady an USSR*. Vol. 269. 1983, pp. 543–547.

[8] Adam Paszke et al. "Pytorch: An imperative style, high-performance deep learning library". In: *Advances in neural information processing systems* 32 (2019).

[9] Boris T. Polyak. "Some Methods of Speeding Up the Convergence of Iteration Methods". In: *USSR Computational Mathematics and Mathematical Physics* 4.5 (1964), pp. 1–17.

[10] Tim Salimans et al. "Evolution Strategies as a Scalable Alternative to Reinforcement Learning". In: (2017). https://arxiv.org/abs/1703.03864.

[11] Ilya Sutskever et al. “On the Importance of Initialization and Momentum in Deep Learning”. In: *International Conference on Machine Learning*. PMLR. 2013, pp. 1139–1147.

[12] Chiyuan Zhang et al. “Understanding Deep Learning (Still) Requires Rethinking Generalization”. In: *Communications of the ACM* 64.3 (2021), pp. 107–115.