Chapter 7, Part 5: Training and Practicalities for Deep Learning Models

Advanced Topics in Statistical Machine Learning

Tom Rainforth

Hilary 2024

rainforth@stats.ox.ac.uk

Gradient Descent

Using the backpropagation/automatic differentiation techniques from the last lecture yields the loss gradient $\nabla_\theta L_i$ for any datapoint, from which we can construct the empirical risk gradient

$$\nabla_\theta \hat{R}(\theta) = \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta L_i + \lambda \nabla_\theta r(\theta)$$

The simplest approach to optimize the network parameters would now be to just use vanilla **gradient descent** by taking updates

$$\theta_t = \theta_{t-1} - \alpha_t \nabla_\theta \hat{R}(\theta_{t-1})$$

where $\alpha_t > 0$ is a scalar **step–size** that is decreased over time.

Given appropriate control of α_t, this will converge to a **stationary point**[1] of $\hat{R}(\theta)$ in the limit of a large number of steps

[1] This could be a **local** minimum or a **saddle point**

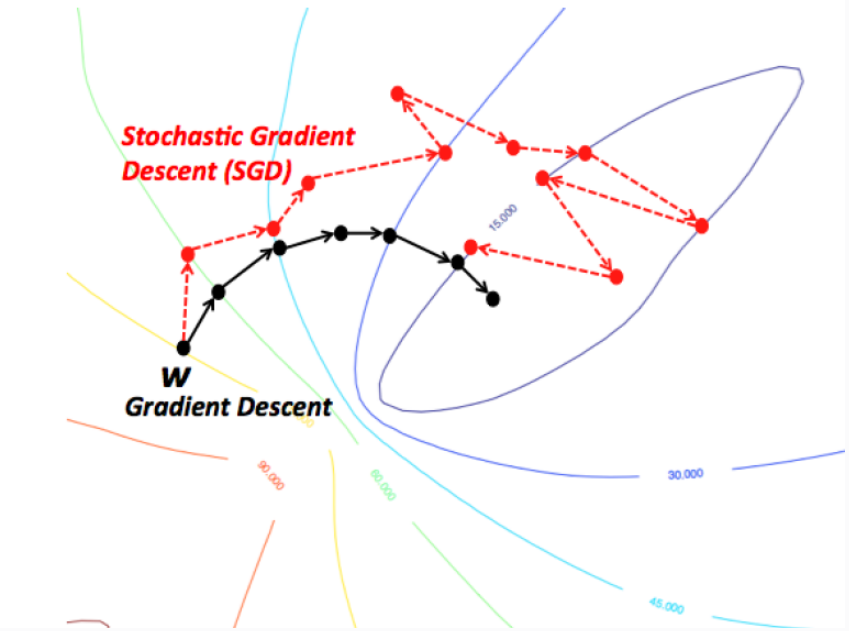
Stochastic Gradients

However, each such step costs $O(n)$ which can very inefficient for large datasets: we do not generally need to ensure we move in **exactly** the direction of steepest descent

The gradient will generally not point **directly** towards the optimum anyway

Taking more **noisy** steps can converge faster than taking fewer more accurate steps

Potential for huge gains for large datasets if we can use steps whose cost is $\ll O(n)$



Source: `https://wikidocs.net/3413`

Stochastic Gradient Descent by Minibatching

We can perform **stochastic gradient descent** (SGD) by taking different randomized **minibatches** of the data $B \subset \{1, \ldots, n\}$ at each iteration and updating using only these datapoints:

$$\theta_t = \theta_{t-1} - \alpha_t \widehat{\nabla_\theta \hat{R}}(\theta_{t-1}) \quad \text{where}$$

$$\widehat{\nabla_\theta \hat{R}}(\theta) = \frac{1}{|B|} \sum_{i \in B} \nabla_\theta L_i + \lambda \nabla_\theta r(\theta)$$

The minibatches are usually generated by randomizing the order of the data and taking the next $|B|$ samples each iteration, looping over the dataset multiple times with each loop known as an **epoch**

The cost of each update is now only $O(|B|)$, which is something we can directly control by choosing the **batch size** $|B|$

Though there is a trade–off (the gradient variance is $O(1/|B|)$) the optimal batch size is usually quite small (and $O(1)$)

Convergence of SGD

For appropriate minibatching schemes, we can view $\widehat{\nabla_\theta \hat{R}(\theta_{t-1})}$ as an **unbiased** Monte Carlo estimate of $\nabla_\theta \hat{R}(\theta_{t-1})$

By linearity, the **expected** value of θ_t given θ_{t-1} for SGD is the same as the deterministic θ_t that would have been produced by using standard gradient descent

This can be used to show that SGD also **converges** to a stationary point of $\hat{R}(\theta)$ (given some weak assumptions) for any batch size if the step sizes satisfy the **Robbins–Monro conditions**:

$$\sum_{t=1}^{\infty} \alpha_t = \infty, \qquad \sum_{t=1}^{\infty} \alpha_t^2 < \infty \tag{1}$$

A traditional choice would be something like $\alpha_t = a/(b+t)$ for positive constants a and b

Momentum

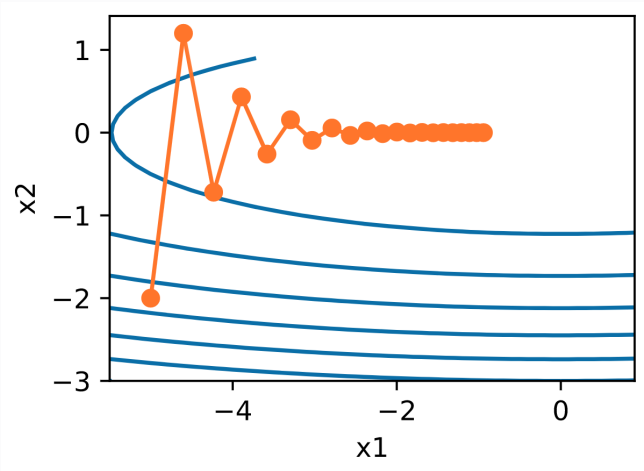
Vanilla SGD can be susceptible to moving extremely slowly through "valleys" where the optimization problem is ill–conditioned

Momentum mitigates this by averaging over previous minibatches:

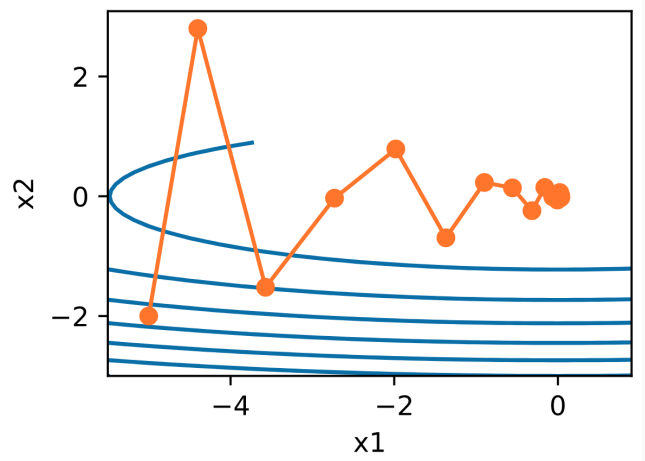
$$\Delta_t = \beta \Delta_{t-1} + (1-\beta)\widehat{\nabla_\theta \hat{R}(\theta_{t-1})}, \qquad 0 < \beta < 1$$

$$\theta_t = \theta_{t-1} - \alpha_t \Delta_t$$

This can also help to reduce noise and allow larger learning rates



Standard SGD



SGD with momentum

Credit: Dive into Deep Learning `https://d2l.ai/index.html`

ADAM

Standard SGD uses a single step size for all parameters, but parameters may have different scales and need different step sizes

ADAM is a popular optimiser that accounts for this by keeping weighted moving average estimates for both the gradient (Δ_t) and the **squared** gradient (V_t), scaling the step size using the latter

$$\Delta_t = \beta_1 \Delta_{t-1} + (1-\beta_1)\widehat{\nabla_\theta \hat{R}}(\theta_{t-1}) \qquad \tilde{\Delta}_t = \frac{\Delta_t}{1-(\beta_1)^t}$$

$$V_t = \beta_2 V_{t-1} + (1-\beta_2)\left(\widehat{\nabla_\theta \hat{R}}(\theta_{t-1})\right)^2 \qquad \tilde{V}_t = \frac{V_t}{1-(\beta_2)^t}$$

$$\theta_t = \theta_{t-1} - \frac{\alpha_t}{\sqrt{\tilde{V}_t} + \epsilon}\tilde{\Delta}_t$$

V_t is related to the curvature of the loss landscape.

Both Δ_0 and V_0 are initialized at 0

Initialization

In general, parameters are initialized **randomly**, typically with **zero-mean** Gaussian distributions

Randomization is important to break symmetry so that each unit in a layer will learn differently

Variances are chosen to ensure that the **pre–activations** are in sensible regions that avoid saturation of the activation function (typically on the order of $[-2, 2]$, e.g. $\tanh$ or sigmoid)

A linear layer causes the pre–activations to be $O(\sigma_\ell \sqrt{d_{\ell-1}})$ times the activations of the previous layer if the weights $\sim \mathcal{N}(0, \sigma_\ell^2)$

We can avoid blow up or collapse by setting $\sigma_\ell^2 = O(1/d_{\ell-1})$

Using $\sigma_\ell^2 = 1/d_{\ell-1}$ exactly is known as **Xavier initialization**

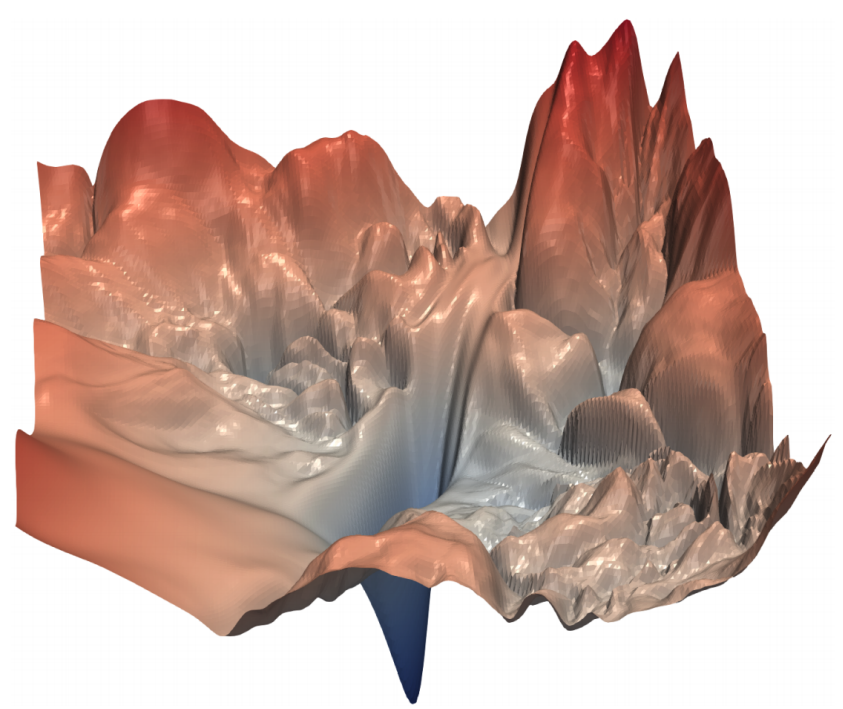
The Loss Landscape

$\hat{R}(\theta)$ is a very high–dimensional and highly non–convex function

We are only using **local** optimizers

However, we rarely reach true local optima: we tend to get stuck in regions where progress is difficult (or find non-unique global optima)

Though deep models tend to be more variable over random seeds than most other ML approaches, they are still remarkably consistent given we only use local optimization

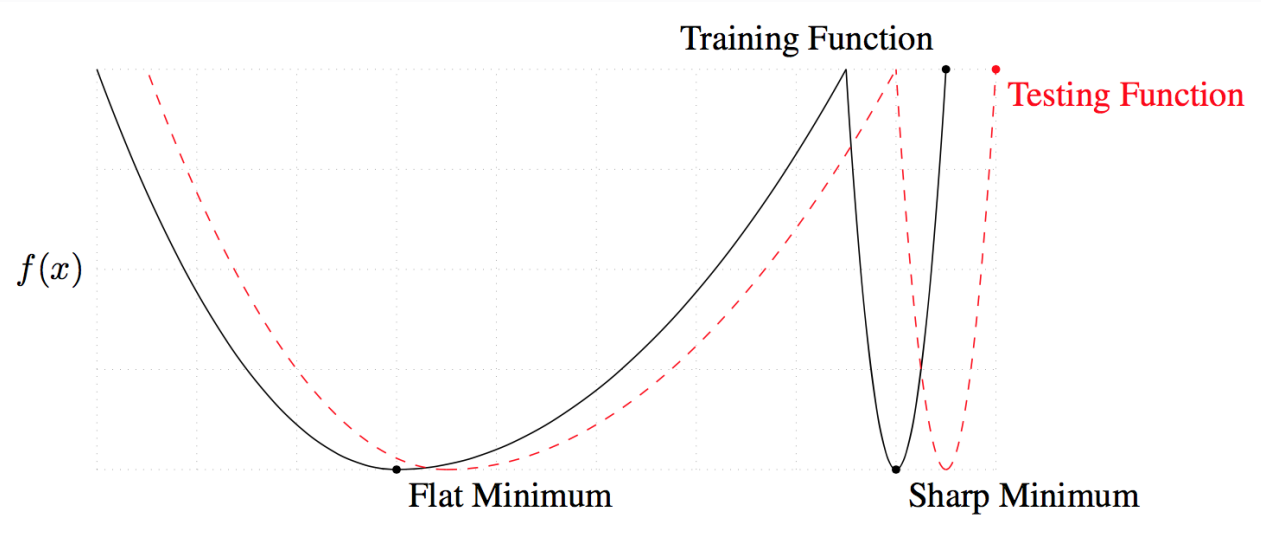


Credit: Visualizing the Loss Landscape of Neural Nets, Li et al, NeurIPS 2018

Regularizing Effects of SGD

The noise in SGD can actually sometimes have helpful **regularizing** effects by avoiding narrow (but potentially deep) local minima that will typically generalize poorly

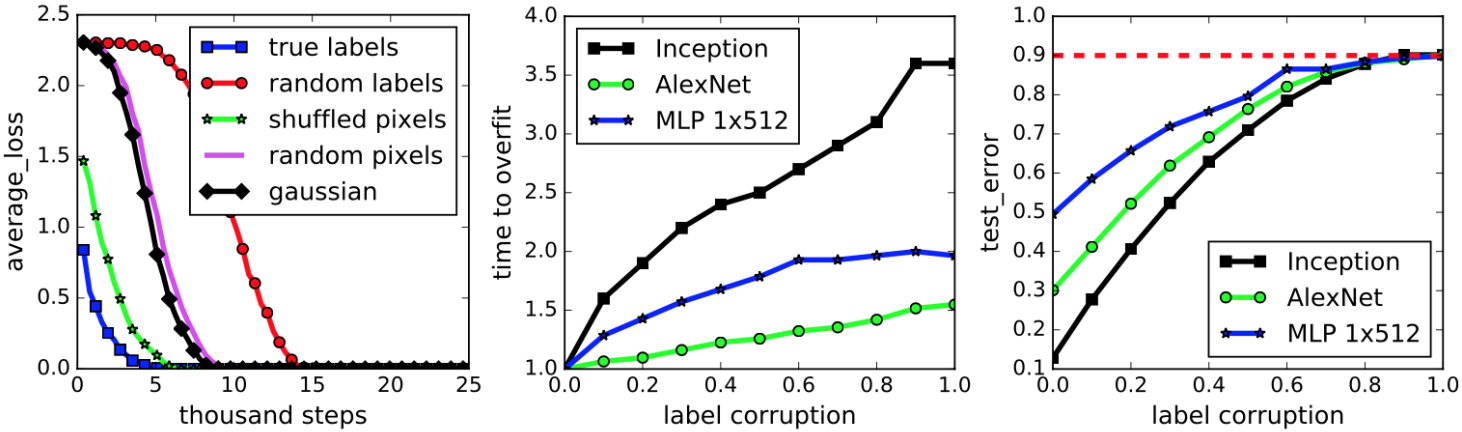
However, this can require step size adaptations that are quite different to those required by the Robbins–Monro conditions (e.g. keeping the step size quite high before using sudden drops)



Source: `https://arxiv.org/abs/1609.04836`

Modern Networks Can Always Overfit

Modern deep neural networks can effectively always overfit any training data if the network is sufficiently large, even producing zero training error on random datasets



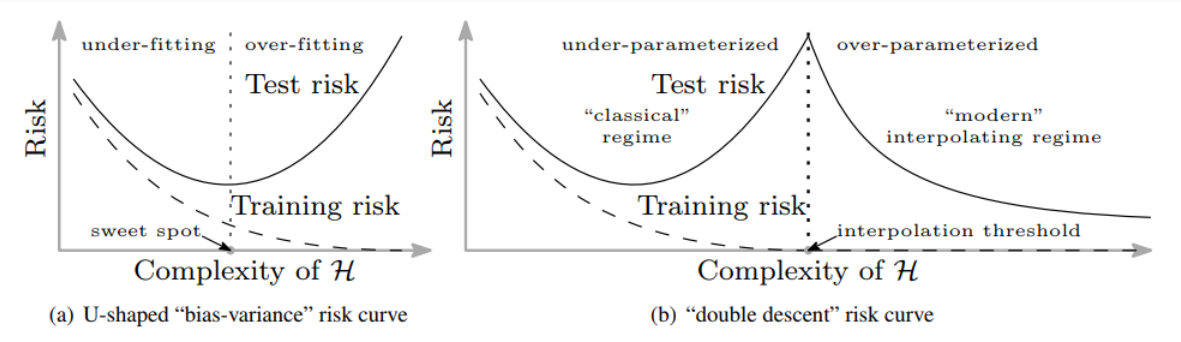
(a) learning curves (b) convergence slowdown (c) generalization error growth

Figure 1: Fitting random labels and random pixels on CIFAR10. (a) shows the training loss of various experiment settings decaying with the training steps. (b) shows the relative convergence time with different label corruption ratio. (c) shows the test error (also the generalization error since training error is 0) under different label corruptions.

Credit: Understanding deep learning requires rethinking generalization. Zhang et al. ICLR, 2017

Double Descent

A strange phenomenon in deep learning is that overparameterized models typically give the best generalization performance



Credit: Belkin et al 2019 (see notes) and OpenAI blog on "Deep Double Descent"

Regularization

The most common (but still rarely used) classical regularization approach used in deep learning is L2 regularization, $r(\theta) = \frac{1}{2}\|\theta\|_2^2$

This is often known as **weight decay** as it equates to replacing θ_{t-1} with $\theta_{t-1}(1 - \lambda\alpha_t)$ at each iteration, such that it encourages a decay towards 0

Instead we typically use **algorithmic** approaches to regularization:

- **Dropout**: randomly remove hidden units from network at each training iteration by temporarily fixing each activation to zero with some probability p (typically $p = 0.5$), sampling the units to be "dropped out" independently at each iteration
- **Early Stopping**: calculate the validation loss for the current network configuration at regular intervals and stop training when it starts increasing.

Computational Considerations

- The bottleneck computation involved in training is typically the large tensor products in the forward pass/backpropagation
- Graphics processing units (GPUs) are typically far more efficient at these than CPUs, sometimes by up to $100+$ times
- Use of GPUs has thus become essential to the practical training of large, and even medium, sized networks
- Good GPUs tend to be very expensive and energy hungry: access to compute has become a bit bottleneck in research as is a driving force in increased industry presence (Google Colab is useful way of getting some free access)
- Modern packages allow for easy transfer of computation onto GPUs, but this must still often be carefully managed
- More recently, custom–built hardware has started to be developed, such as Tensor processing units (TPUs)

A Full Example Implementation in PyTorch

(There is a small error in the following code: you need to change `dataiter.next()` to `next(dataiter)` to make it work)

`https://colab.research.google.com/github/pytorch/tutorials/blob/gh-pages/_downloads/17a7c7cb80916fcdf921097825a0f562/cifar10_tutorial.ipynb`

The Success of Deep Learning

To summarize, the success of deep learning is primarily based on:

- The empirical prowess of large, many–layered, neural networks for problems with a **huge amount** of **high–dimensional** data
- The **flexibility** of the general framework to allow highly customized architectures tailored to specific tasks
- **Automatic differentiation** and deep learning packages making models and their corresponding training schemes easy to construct to run
- Effectiveness of **stochastic gradient schemes** in allowing us to successfully train huge networks
- Suitability of these computations to running on GPUs allowing for big speed ups

Further Reading

- Chapter 11, 12, and 19 of `https://d2l.ai/index.html`
- Chapters 7, 8, and 11 of `https://www.deeplearningbook.org/`
- Andrew Ng on "Nuts and Bolts of Applying Deep Learning" `https://www.youtube.com/watch?v=F1ka6a13S9I`
- Tutorials linked to in previous lectures and extensive help pages for packages like Tensorflow and PyTorch
- Truly massive body of recent work in the literature (non–exhaustive list of prominent publication venues: NeurIPS, ICML, ICLR, JMLR, AISTATS, AAAI, and UAI)