Training Neural Networks

Milan Straka

February 25, 2025

EUROPEAN UNION
European Structural and Investment Fund
Operational Programme Research,
Development and Education

Charles University in Prague
Faculty of Mathematics and Physics
Institute of Formal and Applied Linguistics

Refresh – Neural Networks

- Neural network describes a computation, which gets an input tensor and produces an output.
  - For the time being, the input tensor has a fixed size.
  - The input tensor is usually a vector, but it can be 2D/3D/4D tensor.
    - images, video, time sequences like speech, …
  - The output usually describes a distribution.
    - normal distribution for regression
    - Bernoulli for binary classification
    - categorical for multiclass classification
- The basic units are **nodes**, composed in an acyclic graph.
- The edges have weights, nodes have activation functions.
- Nodes of neural networks are usually composed in layers.

Machine Learning Basics

Machine Learning Basics

We usually have a **training set**, which is assumed to consist of examples generated independently from a **data-generating distribution**.

The goal of *optimization* is to match the training set as well as possible.

However, the goal of *machine learning* is to perform well on *previously unseen* data, to achieve lowest **generalization error** or **test error**. We typically estimate it using a **test set** of examples independent of the training set, but generated by the same data-generating distribution.

The **No free lunch theorem** (Wolpert, 1996) states that averaging over *all possible* data distributions, every classification algorithm achieves the same *overall* error when processing unseen examples (even algorithms "always return 0" and "return the least probable class"). In a sense, no machine learning algorithm is *universally* better than others. *But in practice the data distributions are not uniformly random, so some algorithms might work better in practice than others.*

Machine Learning Basics

Challenges in machine learning:

- *underfitting* (the model is "too weak", bad performance even on training set)
- *overfitting* (the model is "too strong", learned rules are too specific and do not generalize)

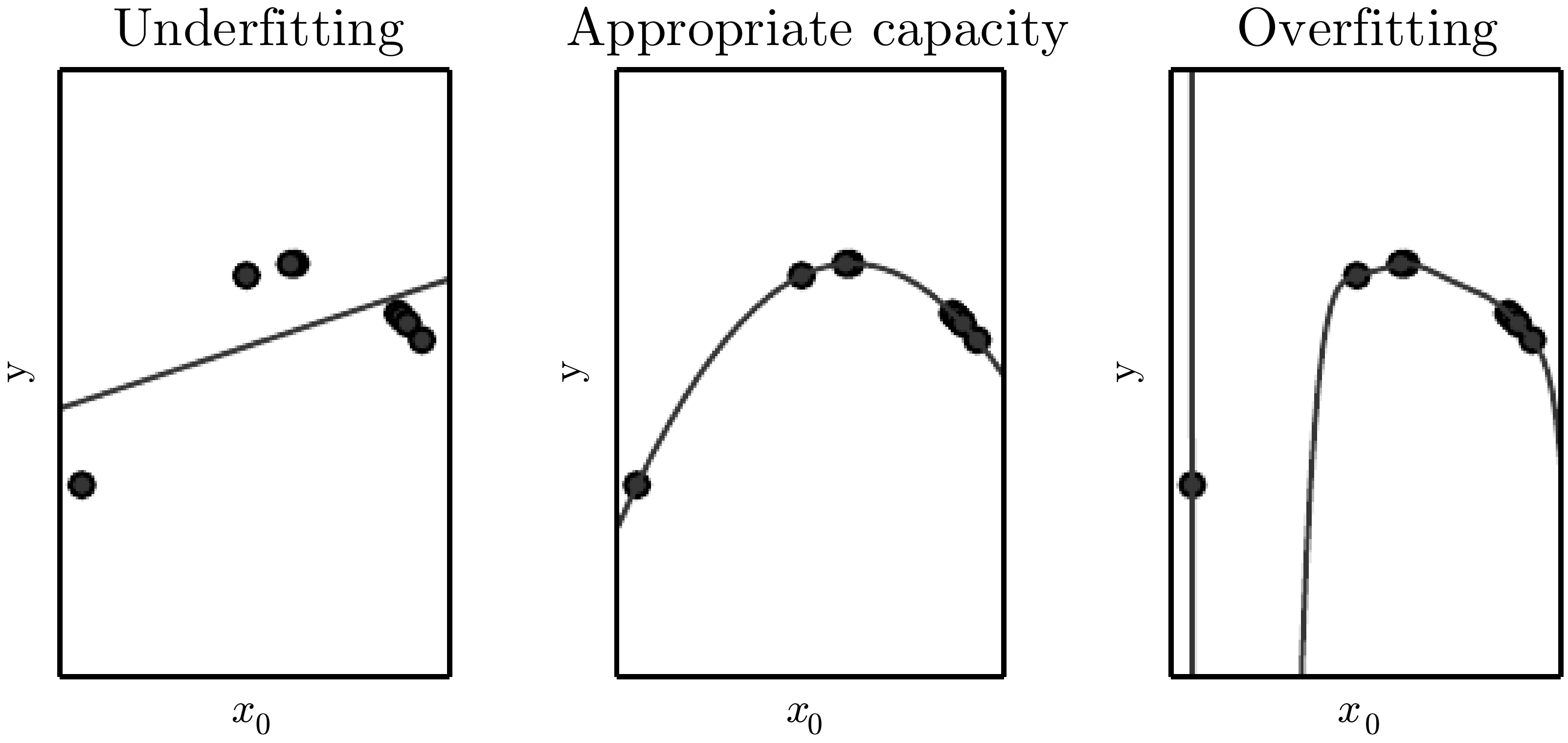
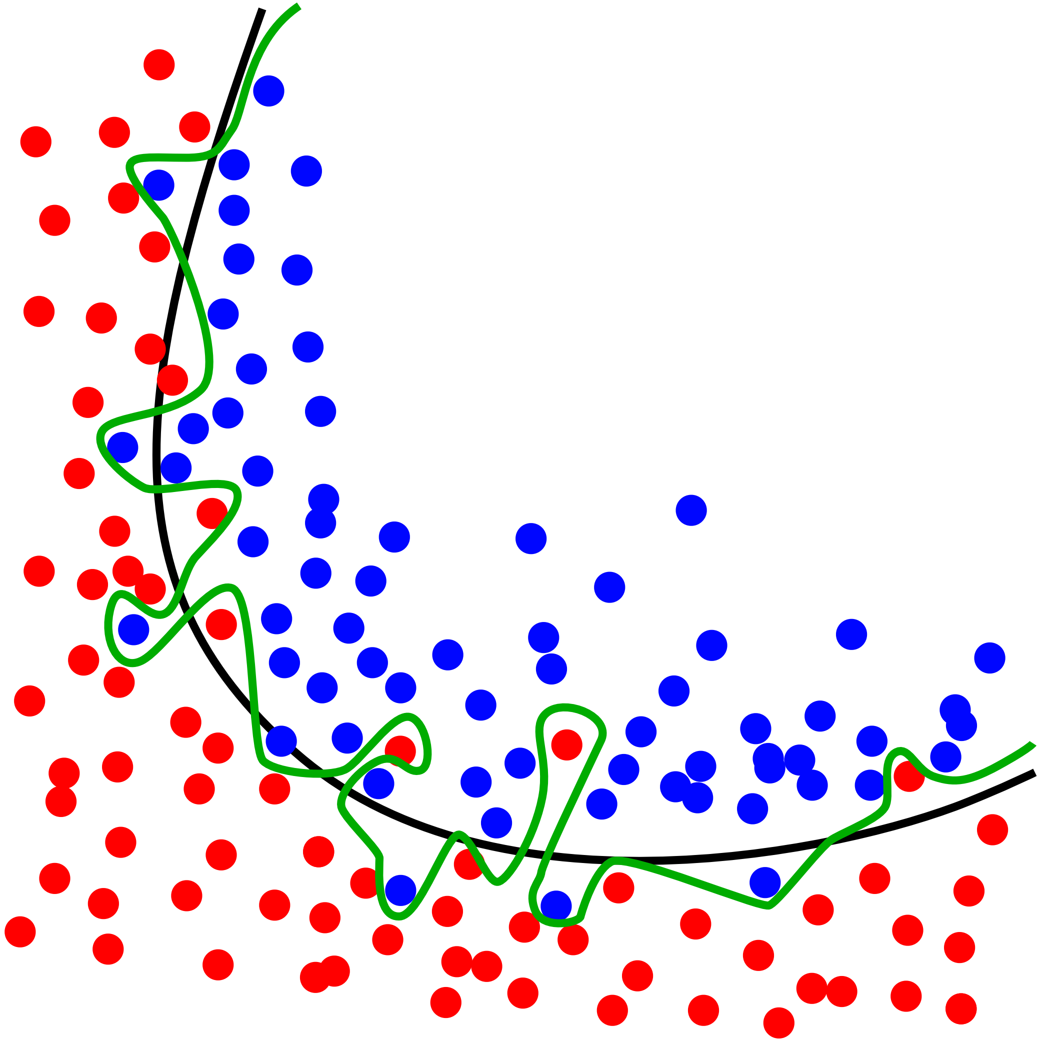


Figure 5.2 of "Deep Learning" book, https://www.deeplearningbook.org

Machine Learning Basics



https://upload.wikimedia.org/wikipedia/commons/1/19/Overfitting.svg

Machine Learning Basics

We can control whether a model underfits or overfits by modifying its *capacity*.

- *representational capacity* (what the model could represent, depends on the model size)
- *effective capacity* (what the model actually learns, depends on training, regularization, …)

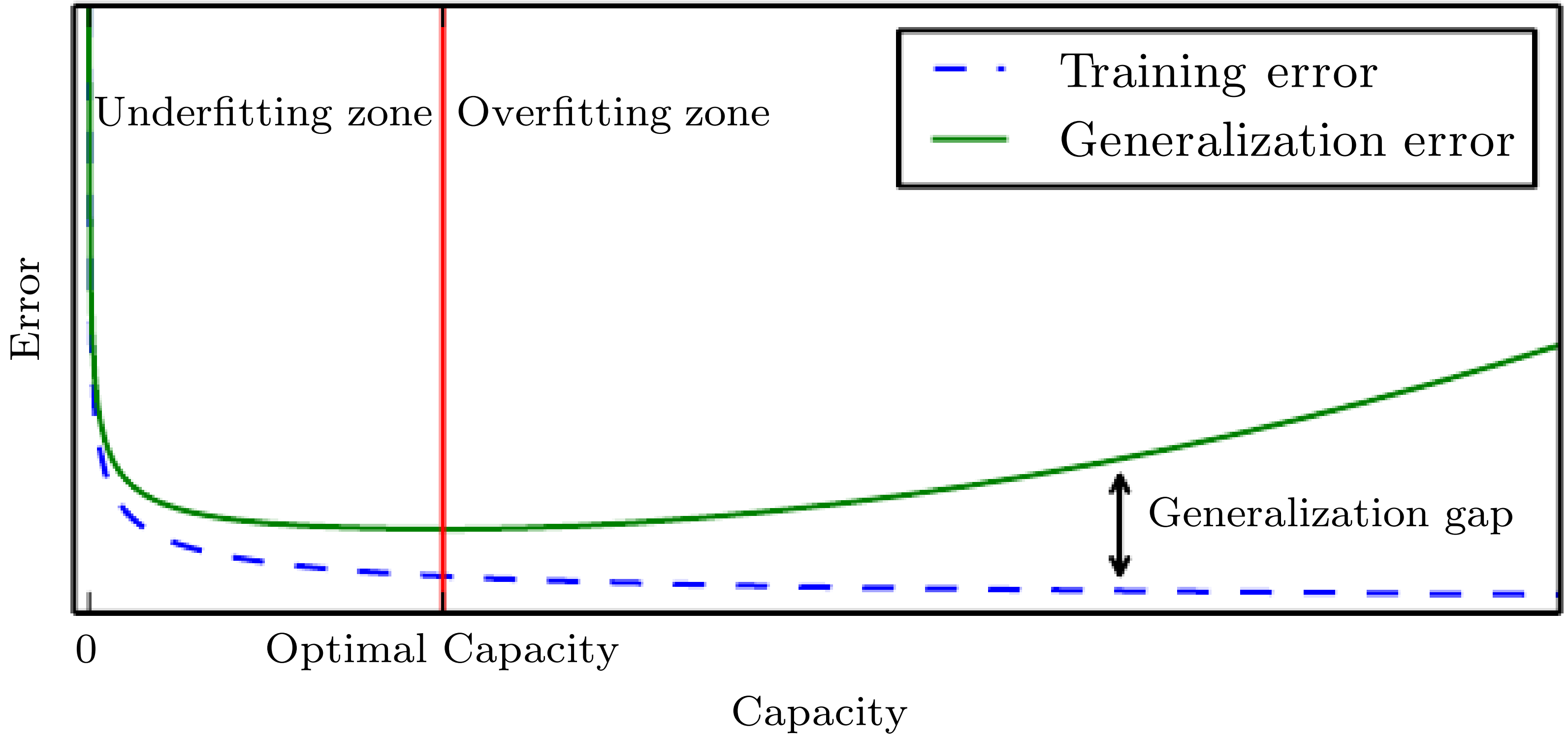


Figure 5.3 of "Deep Learning" book, https://www.deeplearningbook.org

Machine Learning Basics

Overfitting usually decreases with the amount of the training data.

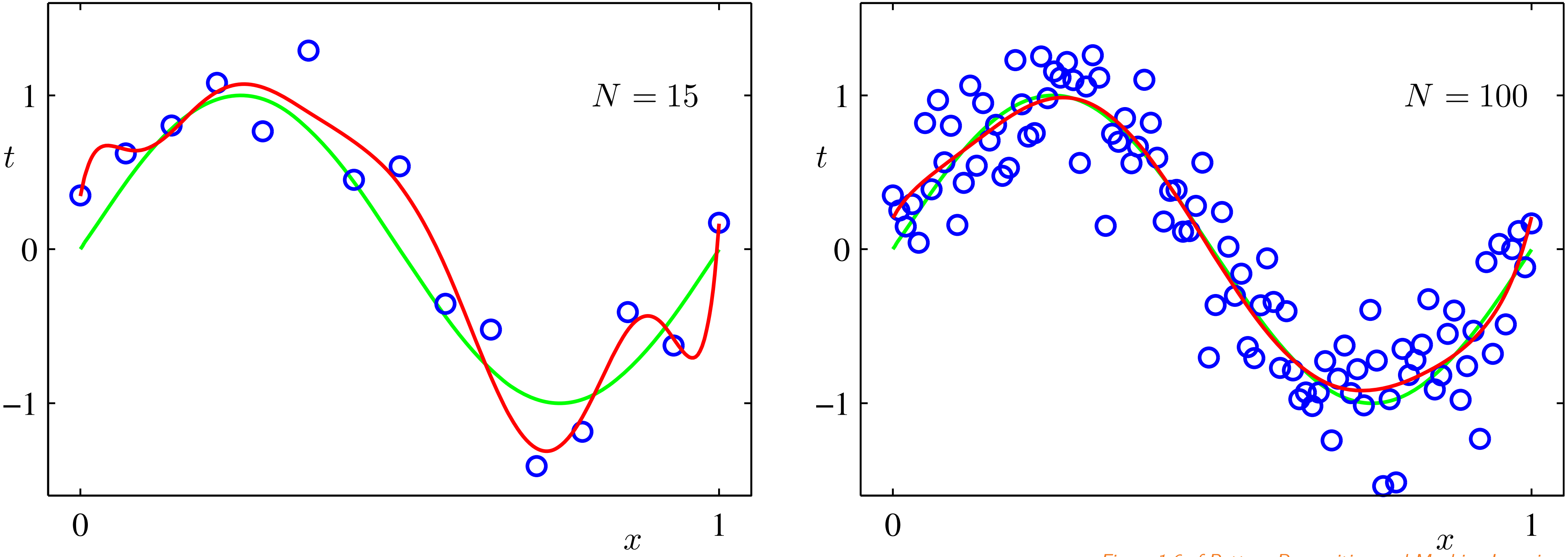


Figure 1.6 of Pattern Recognition and Machine Learning.

Machine Learning Basics

Any change in a machine learning algorithm that is designed to *reduce generalization error* (but not necessarily its training error) is called **regularization**.

L^2 **regularization** (also called **weight decay**) penalizes models with large weights (using a penalty of $\frac{1}{2}\|\boldsymbol{\theta}\|^2$).

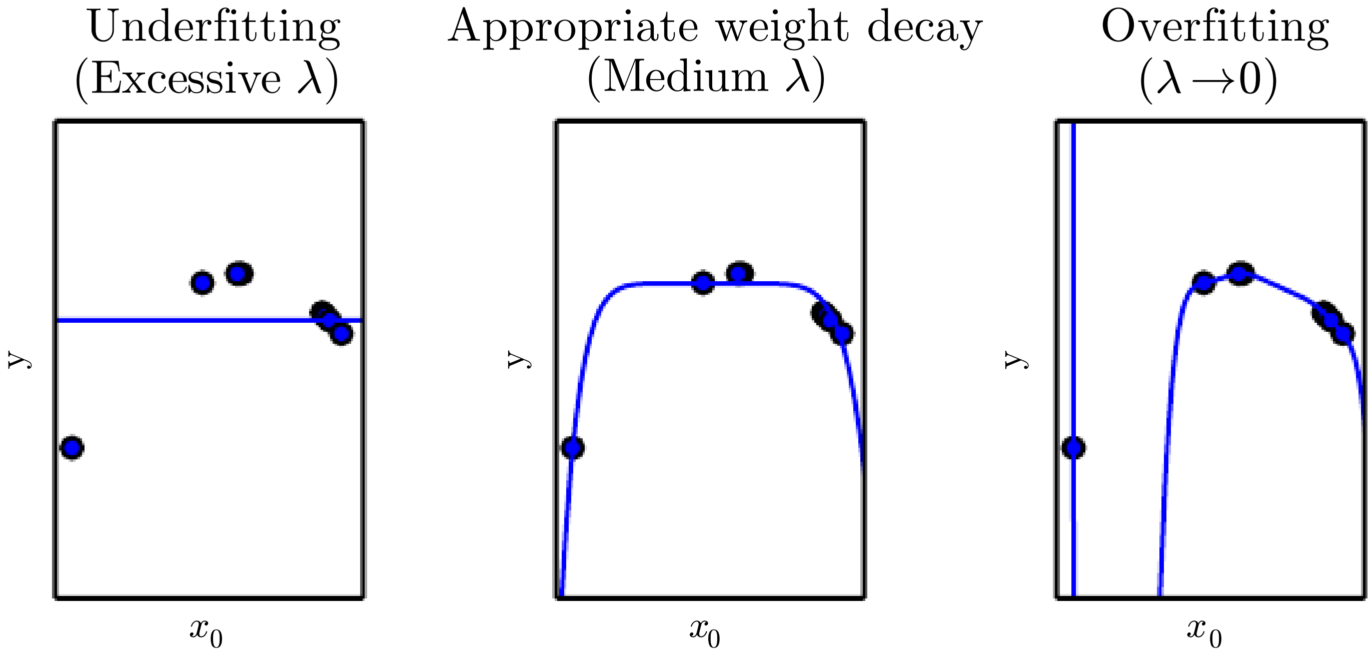


Figure 5.5 of "Deep Learning" book, https://www.deeplearningbook.org

Machine Learning Basics

Hyperparameters are not adapted by a learning algorithm itself, while the model **parameters** (weights, biases) are adapted by it.

Usually a **development set**, also called a **validation set**, is used to estimate the generalization error, allowing to update hyperparameters accordingly.

Maximum Likelihood Estimation

Loss Function

A model is usually trained in order to minimize the **loss** on the training data.

Assuming that a model computes $f(\boldsymbol{x}; \boldsymbol{\theta})$ using parameters $\boldsymbol{\theta}$, the **mean square error** of given N examples $\big(\boldsymbol{x}^{(1)}, y^{(1)}\big), \big(\boldsymbol{x}^{(2)}, y^{(2)}\big), \ldots, \big(\boldsymbol{x}^{(N)}, y^{(N)}\big)$ is computed as

$$\frac{1}{N}\sum_{i=1}^{N}\Big(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}) - y^{(i)}\Big)^2.$$

A common principle used to design loss functions is the **maximum likelihood principle**.

Maximum Likelihood Estimation

Let $\mathbb{X} = \{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(N)}\}$ be training data drawn independently from the data-generating distribution p_data.

We denote the **empirical data distribution** as $\hat p_\text{data}$, where

$$\hat p_\text{data}(\boldsymbol{x}) \stackrel{\text{def}}{=} \frac{\left|\{i : \boldsymbol{x}^{(i)} = \boldsymbol{x}\}\right|}{N}.$$

Let $p_\text{model}(\mathbf{x}; \boldsymbol{\theta})$ be a family of distributions.

- If the weights are fixed, $p_\text{model}(\mathbf{x}; \boldsymbol{\theta})$ is a probability distribution.
- If we instead consider the fixed training data $\mathbb{X}$, then

$$L(\boldsymbol{\theta}) = p_\text{model}(\mathbb{X}; \boldsymbol{\theta}) = \prod\nolimits_{i=1}^N p_\text{model}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta})$$

is called the **likelihood**. Note that even if the value of the likelihood is in range $[0, 1]$, it is not a probability, because the likelihood is not a probability distribution.

Maximum Likelihood Estimation

Let $\mathbb{X} = \{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(N)}\}$ be training data drawn independently from the data-generating distribution $p_{\textrm{data}}$. We denote the empirical data distribution as $\hat{p}_{\textrm{data}}$ and let $p_{\textrm{model}}(\mathbf{x}; \boldsymbol{\theta})$ be a family of distributions.

The **maximum likelihood estimation** of $\boldsymbol{\theta}$ is:

$$\begin{aligned}
\boldsymbol{\theta}_{\textrm{MLE}} &= \arg\max_{\boldsymbol{\theta}} p_{\textrm{model}}(\mathbb{X}; \boldsymbol{\theta}) = \arg\max_{\boldsymbol{\theta}} \prod\nolimits_{i=1}^N p_{\textrm{model}}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}) \\
&= \arg\min_{\boldsymbol{\theta}} \sum\nolimits_{i=1}^N -\log p_{\textrm{model}}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}) \\
&= \arg\min_{\boldsymbol{\theta}} \mathbb{E}_{\mathbf{x} \sim \hat{p}_{\textrm{data}}} [-\log p_{\textrm{model}}(\boldsymbol{x}; \boldsymbol{\theta})] \\
&= \arg\min_{\boldsymbol{\theta}} H(\hat{p}_{\textrm{data}}(\mathbf{x}), p_{\textrm{model}}(\mathbf{x}; \boldsymbol{\theta})) \\
&= \arg\min_{\boldsymbol{\theta}} D_{\textrm{KL}}(\hat{p}_{\textrm{data}}(\mathbf{x}) \| p_{\textrm{model}}(\mathbf{x}; \boldsymbol{\theta})) + H(\hat{p}_{\textrm{data}}(\mathbf{x}))
\end{aligned}$$

Maximum Likelihood Estimation

MLE can be easily generalized to the conditional case, where our goal is to predict y given $\boldsymbol{x}$:

$$\begin{aligned}
\boldsymbol{\theta}_{\mathrm{MLE}} &= \arg\max_{\boldsymbol{\theta}} p_{\text{model}}(\mathbb{Y}|\mathbb{X};\boldsymbol{\theta}) = \arg\max_{\boldsymbol{\theta}} \prod\nolimits_{i=1}^{N} p_{\text{model}}(y^{(i)}|\boldsymbol{x}^{(i)};\boldsymbol{\theta}) \\
&= \arg\min_{\boldsymbol{\theta}} \sum\nolimits_{i=1}^{N} -\log p_{\text{model}}(y^{(i)}|\boldsymbol{x}^{(i)};\boldsymbol{\theta}) \\
&= \arg\min_{\boldsymbol{\theta}} \mathbb{E}_{(\mathbf{x},\mathrm{y})\sim\hat{p}_{\text{data}}}[-\log p_{\text{model}}(y|\boldsymbol{x};\boldsymbol{\theta})] \\
&= \arg\min_{\boldsymbol{\theta}} H(\hat{p}_{\text{data}}(\mathrm{y}|\mathbf{x}), p_{\text{model}}(\mathrm{y}|\mathbf{x};\boldsymbol{\theta})) \\
&= \arg\min_{\boldsymbol{\theta}} D_{\mathrm{KL}}(\hat{p}_{\text{data}}(\mathrm{y}|\mathbf{x})\|p_{\text{model}}(\mathrm{y}|\mathbf{x};\boldsymbol{\theta})) + H(\hat{p}_{\text{data}}(\mathrm{y}|\mathbf{x}))
\end{aligned}$$

where the conditional entropy is defined as $H(\hat{p}_{\text{data}}) = \mathbb{E}_{(\mathbf{x},\mathrm{y})\sim\hat{p}_{\text{data}}}[-\log(\hat{p}_{\text{data}}(y|\boldsymbol{x}))]$ and the conditional cross-entropy as $H(\hat{p}_{\text{data}}, p_{\text{model}}) = \mathbb{E}_{(\mathbf{x},\mathrm{y})\sim\hat{p}_{\text{data}}}[-\log(p_{\text{model}}(y|\boldsymbol{x};\boldsymbol{\theta}))]$.

The resulting *loss function* is called **negative log-likelihood (NLL)**, or **cross-entropy**, or **Kullback-Leibler divergence**.

Estimators and Bias

An **estimator** is a rule for computing an estimate of a given value, often an expectation of some random value(s). For example, we might estimate *mean* of a random variable by sampling a value according to its probability distribution.

The **bias** of an estimator is the difference of the expected value of the estimator and the true value being estimated. If the bias is zero, we call the estimator **unbiased**, otherwise **biased**.

If we have a sequence of estimates, it might also happen that the bias converges to zero. Consider the well-known sample estimate of variance. Given independent and identically distributed random variables $\mathbf{x}_1, \ldots, \mathbf{x}_N$, we might estimate the mean and the variance as

$$\hat\mu = \frac{1}{N} \sum\nolimits_i x_i, \quad \hat\sigma^2 = \frac{1}{N} \sum\nolimits_i (x_i - \hat\mu)^2.$$

Such a mean estimate is unbiased, but the estimate of the variance is biased, because $\mathbb{E}[\hat\sigma^2] = (1 - \frac{1}{N})\sigma^2$; however, the bias of this estimate converges to zero for increasing N.

Also, an unbiased estimator does not necessarily have a small variance – in some cases, it can have a large variance, so a biased estimator with a smaller variance might be preferred.

Properties of Maximum Likelihood Estimation

Assume that the true data-generating distribution p_data lies within the model family $p_\text{model}(\bullet; \boldsymbol\theta)$, and assume there exists a unique $\boldsymbol\theta_{p_\text{data}}$ such that $p_\text{data} = p_\text{model}(\bullet; \boldsymbol\theta_{p_\text{data}})$.

- MLE is a *consistent* estimator. If we denote $\boldsymbol\theta_m$ to be the parameters found by MLE for a training set with m examples generated by the data-generating distribution, then $\boldsymbol\theta_m$ converges in probability to $\boldsymbol\theta_{p_\text{data}}$.

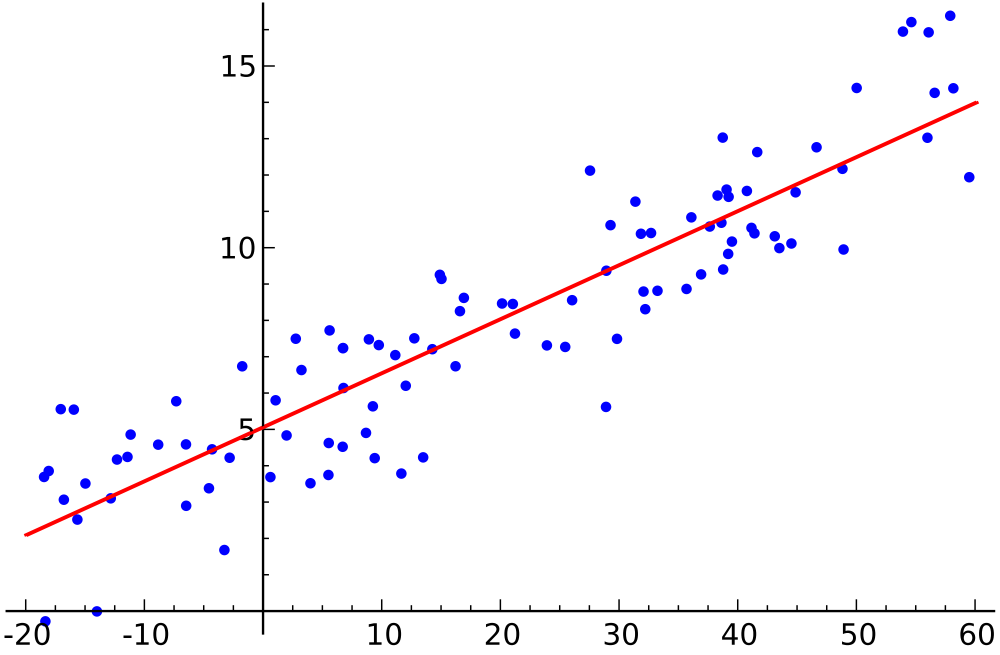
  Formally, for any $\varepsilon > 0$, $P(\|\boldsymbol\theta_m - \boldsymbol\theta_{p_\text{data}}\| > \varepsilon) \rightarrow 0$ as $m \rightarrow \infty$.

- MLE is in a sense the *most statistically efficient*. For any consistent estimator, let us consider the average distance of $\boldsymbol\theta_m$ and $\boldsymbol\theta_{p_\text{data}}$: $\mathbb{E}_{\mathbf{x}_1,\ldots,\mathbf{x}_m \sim p_\text{data}}\left[\|\boldsymbol\theta_m - \boldsymbol\theta_{p_\text{data}}\|^2\right]$.
  It can be shown (Rao 1945, Cramér 1946) that no consistent estimator has lower mean squared error than the maximum likelihood estimator.

Therefore, for reasons of consistency and efficiency, maximum likelihood is often considered the preferred estimator for machine learning.

Mean Square Error as MLE

During regression, we predict a number, not a real probability distribution. In order to generate a distribution, we might consider a distribution with the mean of the predicted value and a fixed variance σ^2 – the most general such a distribution is the normal distribution.



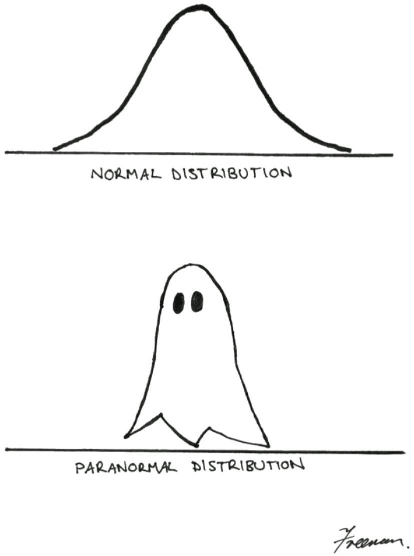
https://upload.wikimedia.org/wikipedia/commons/3/3a/Linear_regression.svg

Mean Square Error as MLE

Let $f(\boldsymbol{x};\boldsymbol{\theta})$ be the output of our model, which we assume to be the mean of y.

We define $p(y|\boldsymbol{x};\boldsymbol{\theta})$ as $\mathcal{N}(y; f(\boldsymbol{x};\boldsymbol{\theta}), \sigma^2)$ for some fixed σ^2. The MLE then results in

$$\begin{aligned}
\arg\max_{\boldsymbol{\theta}} p(\mathbb{Y}|\mathbb{X};\boldsymbol{\theta}) &= \arg\min_{\boldsymbol{\theta}} \sum_{i=1}^N -\log p(y^{(i)}|\boldsymbol{x}^{(i)};\boldsymbol{\theta}) \\
&= \arg\min_{\boldsymbol{\theta}} -\sum_{i=1}^N \log \sqrt{\frac{1}{2\pi\sigma^2}} e^{-\frac{(y^{(i)}-f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}))^2}{2\sigma^2}} \\
&= \arg\min_{\boldsymbol{\theta}} -N\log(2\pi\sigma^2)^{-1/2} - \sum_{i=1}^N -\frac{\left(y^{(i)} - f(\boldsymbol{x}^{(i)};\boldsymbol{\theta})\right)^2}{2\sigma^2} \\
&= \arg\min_{\boldsymbol{\theta}} \sum_{i=1}^N \frac{\left(y^{(i)} - f(\boldsymbol{x}^{(i)};\boldsymbol{\theta})\right)^2}{2\sigma^2} = \arg\min_{\boldsymbol{\theta}} \frac{1}{N} \sum_{i=1}^N \left(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}) - y^{(i)}\right)^2.
\end{aligned}$$



https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2465539/

Gradient Descent

Gradient Descent

Let $f(\boldsymbol x; \boldsymbol\theta)$ be a model with parameters $\boldsymbol\theta$. For a given per-example loss function L, denote

$$E(\boldsymbol\theta) = \mathbb{E}_{(\mathbf{x},\mathrm{y})\sim\hat p_\textrm{data}} L\big(f(\boldsymbol x; \boldsymbol\theta), y\big).$$

Assuming we are minimizing a loss function

$$\argmin_{\boldsymbol\theta} E(\boldsymbol\theta),$$

we may use *gradient descent*:

$$\boldsymbol\theta \leftarrow \boldsymbol\theta - \alpha\nabla_{\boldsymbol\theta} E(\boldsymbol\theta).$$

The constant α is called a **learning rate** and specifies the "length" of a step we perform in every iteration of the gradient descent.

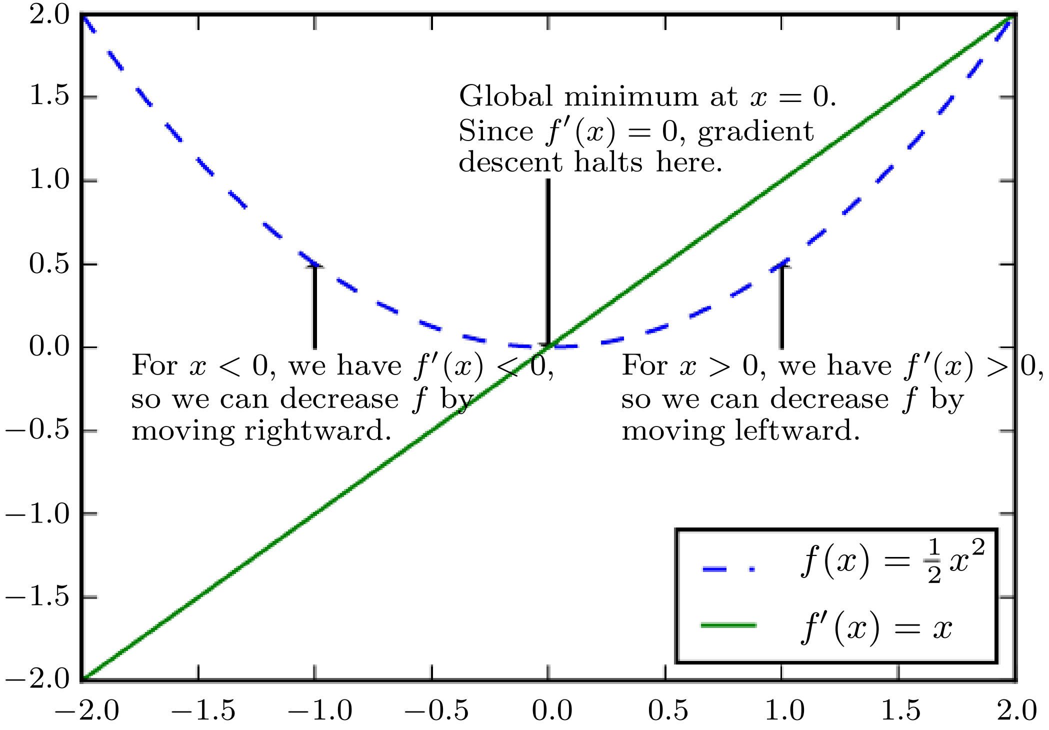


Figure 4.1 of "Deep Learning" book, https://www.deeplearningbook.org

Gradient Descent Variants

The gradient of the loss function $E(\boldsymbol\theta)$ can be computed as

$$\nabla_{\boldsymbol\theta} E(\boldsymbol\theta) = \mathbb{E}_{(\mathbf{x},\mathrm{y})\sim\hat p_\textrm{data}} \nabla_{\boldsymbol\theta} L\big(f(\boldsymbol x; \boldsymbol\theta), y\big).$$

- **(Standard/Batch) Gradient Descent**: We use all training data to compute $\nabla_{\boldsymbol\theta} E(\boldsymbol\theta)$.
- **Stochastic (or Online) Gradient Descent**: We estimate $\nabla_{\boldsymbol\theta} E(\boldsymbol\theta)$ using a single random example from the training data. Such an estimate is unbiased, but very noisy.

$$\nabla_{\boldsymbol\theta} E(\boldsymbol\theta) \approx \nabla_{\boldsymbol\theta} L\big(f(\boldsymbol x; \boldsymbol\theta), y\big) \textrm{ for a randomly chosen } (\boldsymbol x, y) \textrm{ from } \hat p_\textrm{data}.$$

- **Minibatch SGD**: Trade-off between gradient descent and SGD – the expectation in $\nabla_{\boldsymbol\theta} E(\boldsymbol\theta)$ is estimated using m random independent examples from the training data.

$$\nabla_{\boldsymbol\theta} E(\boldsymbol\theta) \approx \frac{1}{m}\sum_{i=1}^m \nabla_{\boldsymbol\theta} L\big(f(\boldsymbol x^{(i)}; \boldsymbol\theta), y^{(i)}\big) \textrm{ for randomly chosen } (\boldsymbol x^{(i)}, y^{(i)}) \textrm{ from } \hat p_\textrm{data}.$$

Stochastic Gradient Descent Convergence

Assume that we perform a stochastic gradient descent, using a sequence of learning rates α_i, and using a noisy estimate $J(\boldsymbol\theta)$ of the real gradient $\nabla_{\boldsymbol\theta} E(\boldsymbol\theta)$:

$$\boldsymbol\theta_{i+1} \leftarrow \boldsymbol\theta_i - \alpha_i J(\boldsymbol\theta_i).$$

It can be proven (see Robbins and Monro algorithm, 1951) that if $\big|J(\boldsymbol\theta)\big|$ is bounded and the loss function is strictly convex and continuous, then SGD converges to the unique optimum almost surely if the sequence of learning rates α_i fulfills the following conditions:

$$\underbrace{\sum_i \alpha_i = \infty,}_{\text{we can travel any distance in the loss landscape,}} \qquad \underbrace{\sum_i \alpha_i^2 < \infty.}_{\text{the learning rates decrease sufficiently quickly}}$$

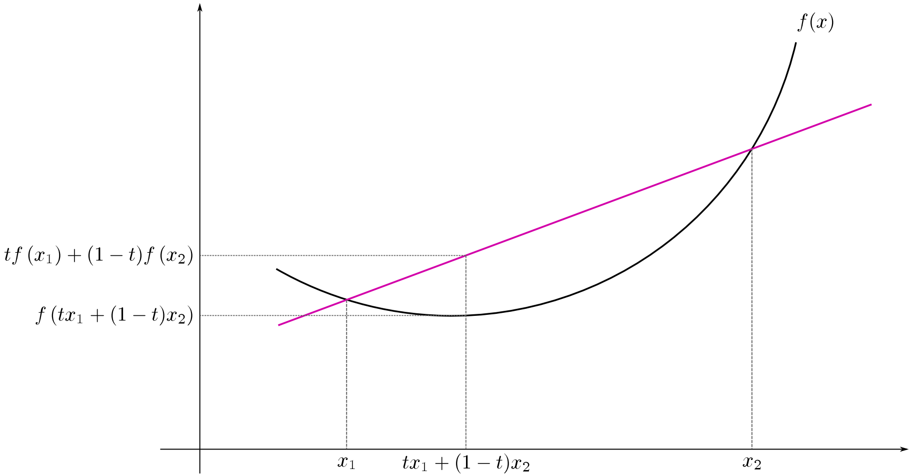
Note that the second condition implies that $\alpha_i \to 0$.

For nonconvex loss functions, we can get guarantees of converging to a *local* optimum only. However, note that finding the global minimum of even a boolean function is *at least NP-hard*.

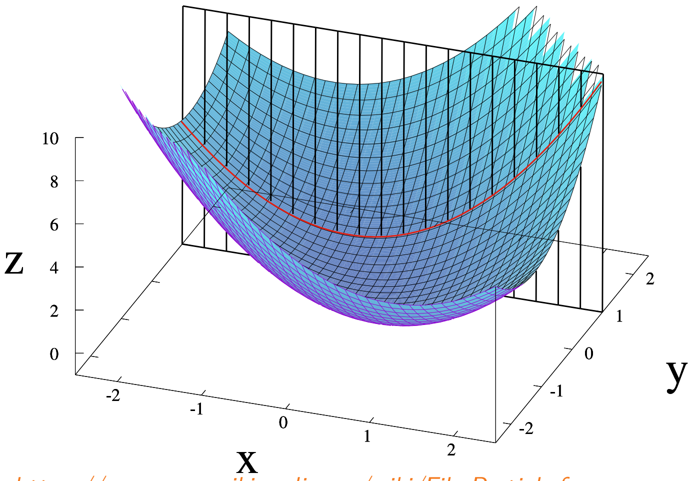
Stochastic Gradient Descent Convergence

Convex functions mentioned on the previous slide are such that for $\boldsymbol{u}, \boldsymbol{v}$ and real $0 \leq t \leq 1$,

$$f(t\boldsymbol{u} + (1-t)\boldsymbol{v}) \leq tf(\boldsymbol{u}) + (1-t)f(\boldsymbol{v}).$$



https://upload.wikimedia.org/wikipedia/commons/c/c7/ConvexFunction.svg



https://commons.wikimedia.org/wiki/File:Partial_func_eg.svg

A twice-differentiable function of a single variable is convex iff its second derivative is always nonnegative. (For functions of multiple variables, the Hessian must be positive semi-definite.)

A local minimum of a convex function is always a global minimum.

Well-known examples of convex functions are x^2, e^x, $-\log x$, MSE, σ+NLL, $\mathrm{softmax}$+NLL.

Loss Function Visualization

Visualization of loss function of ResNet-56 (0.85 million parameters) with/without skip connections:

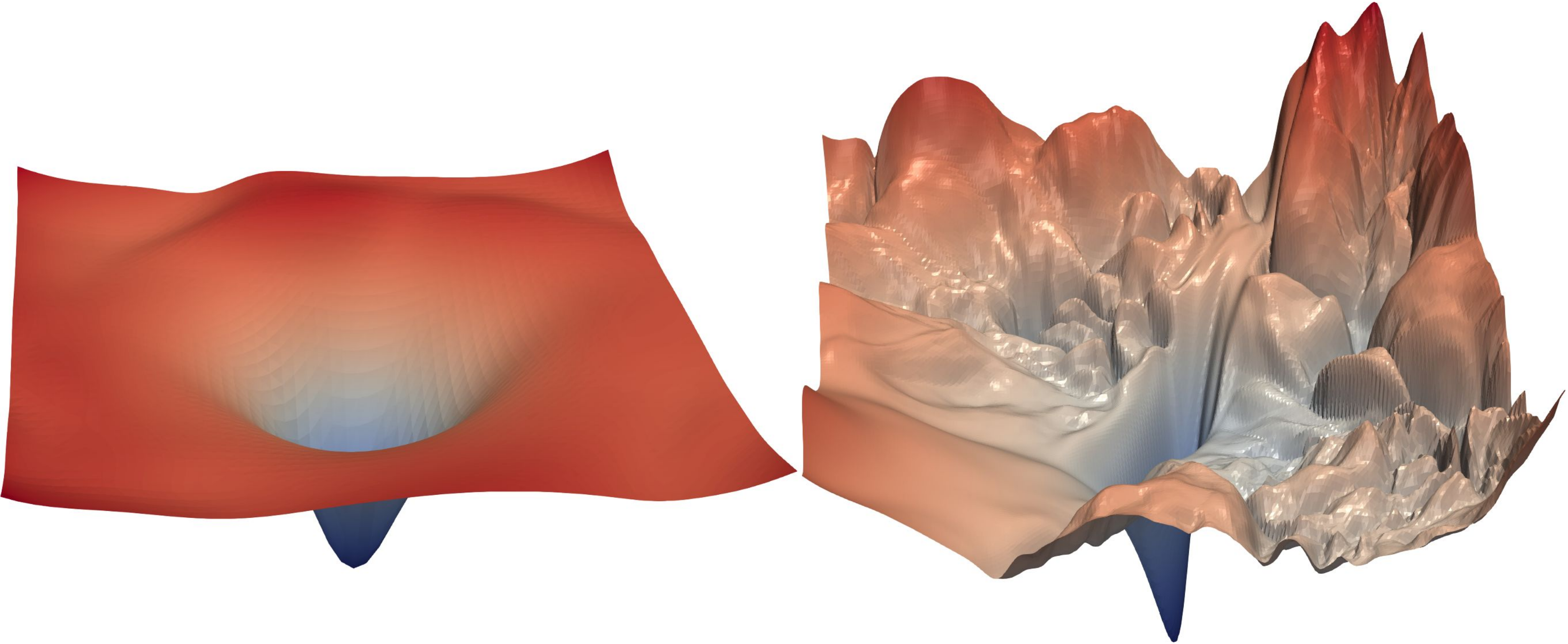


Figure 1 of "Visualizing the Loss Landscape of Neural Nets", https://arxiv.org/abs/1712.09913

Loss Function Visualization

Visualization of loss function of ResNet-110 without skip connections and DenseNet-121:

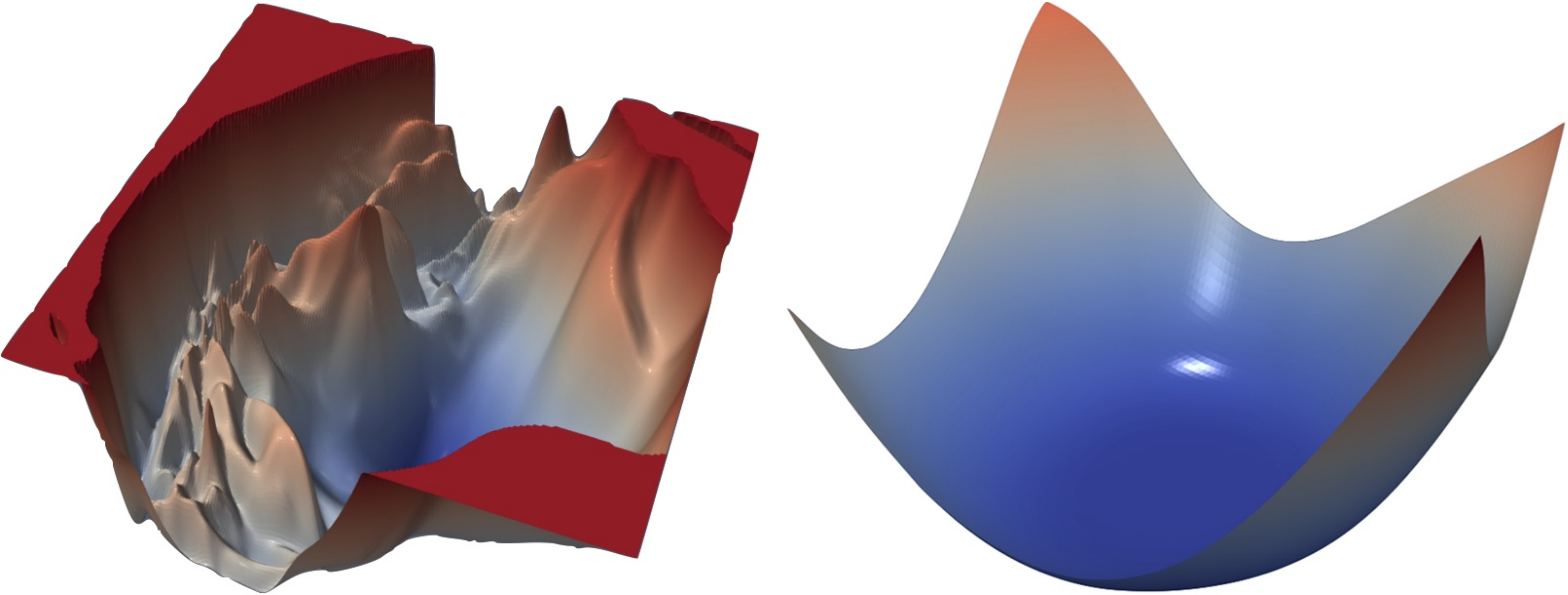
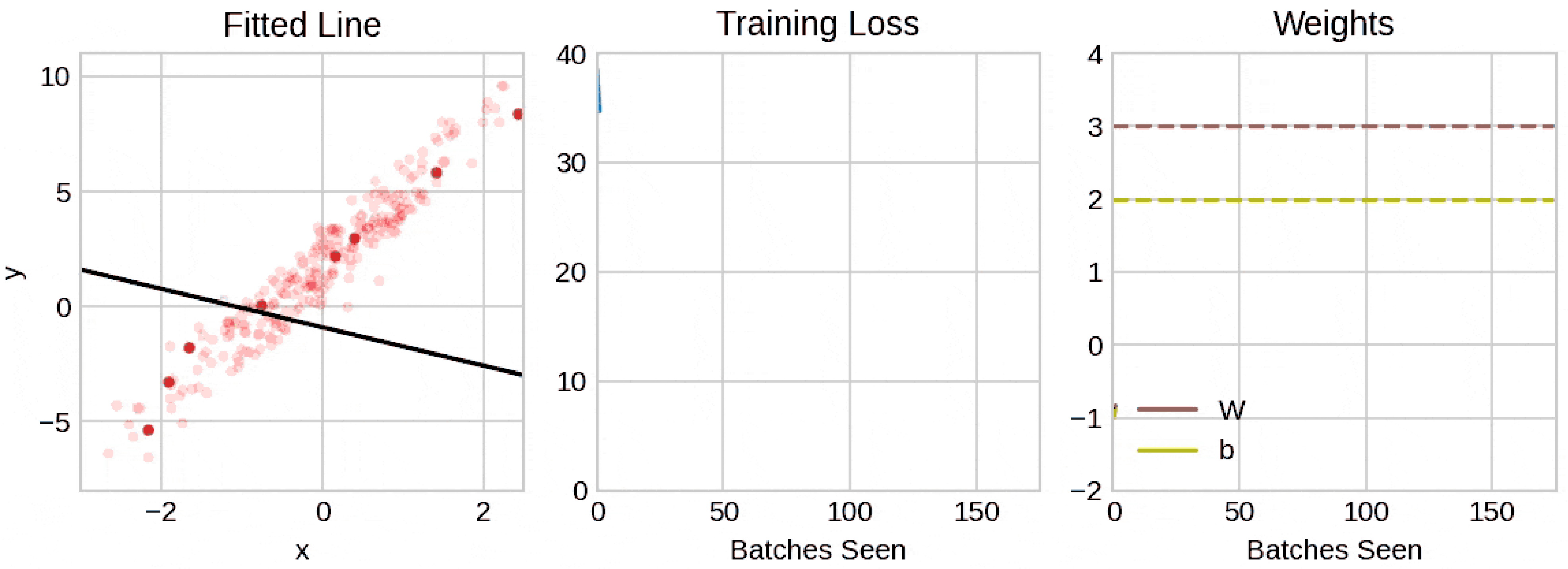


Figure 4 of "Visualizing the Loss Landscape of Neural Nets", https://arxiv.org/abs/1712.09913

Stochastic Gradient Descent Visualization

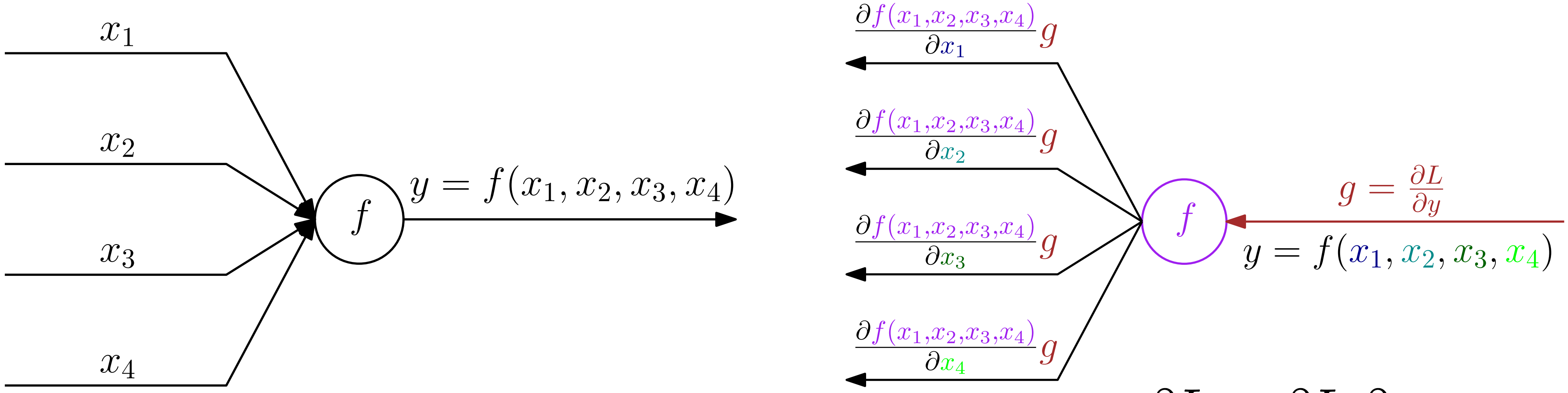
You can explore the interactive figures 2.1, 2.2, 2.4 at https://udlbook.github.io/udlfigures/.



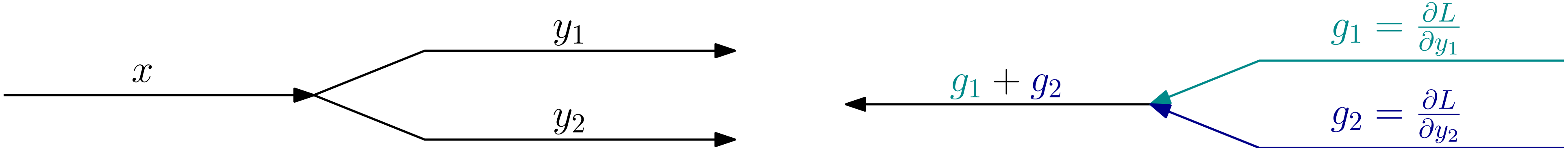
https://www.kaggle.com/code/ryanholbrook/stochastic-gradient-descent

Backpropagation

Assume we want to compute partial derivatives of a given loss function L.


x_1
x_2
x_3
x_4
f
$y = f(x_1, x_2, x_3, x_4)$
$\frac{\partial f(x_1,x_2,x_3,x_4)}{\partial x_1} g$
$\frac{\partial f(x_1,x_2,x_3,x_4)}{\partial x_2} g$
$\frac{\partial f(x_1,x_2,x_3,x_4)}{\partial x_3} g$
$\frac{\partial f(x_1,x_2,x_3,x_4)}{\partial x_4} g$
f
$g = \frac{\partial L}{\partial y}$
$y = f(x_1, x_2, x_3, x_4)$

The gradient computation is based on the chain rule of derivatives: $\dfrac{\partial L}{\partial x_i} = \dfrac{\partial L}{\partial y} \dfrac{\partial y}{\partial x_i}$.


x
y_1
y_2
$g_1 + g_2$
$g_1 = \frac{\partial L}{\partial y_1}$
$g_2 = \frac{\partial L}{\partial y_2}$

Forward Propagation Algorithm

Forward Propagation

Input: Network with nodes $u^{(1)}, u^{(2)}, \ldots, u^{(n)}$ numbered in topological order.
Each node's value is computed as $u^{(i)} = f^{(i)}(A^{(i)})$ for $A^{(i)}$ being a set of values of the predecessors $P(u^{(i)})$ of $u^{(i)}$.
Output: Value of $u^{(n)}$.

```
• For $i = 1, \ldots, n$:
    ○ $A^{(i)} \leftarrow \{u^{(j)} | j \in P(u^{(i)})\}$
    ○ $u^{(i)} \leftarrow f^{(i)}(A^{(i)})$
• Return $u^{(n)}$
```

Backpropagation Algorithm

Simple Variant of Backpropagation

Input: The network as in the Forward propagation algorithm.
Output: Partial derivatives $g^{(i)} = \frac{\partial u^{(n)}}{\partial u^{(i)}}$ of $u^{(n)}$ with respect to all $u^{(i)}$.

- Run forward propagation to compute all $u^{(i)}$
- $g^{(n)} = 1$
- For $i = n-1, \ldots, 1$:
  - $g^{(i)} \leftarrow \sum_{j: i \in P(u^{(j)})} g^{(j)} \frac{\partial u^{(j)}}{\partial u^{(i)}}$
- Return $\left(g^{(1)}, g^{(2)}, \ldots, g^{(n)}\right)$

In practice, we do not usually represent networks as collections of scalar nodes; instead we represent them as collections of tensor functions – most usually functions $f: \mathbb{R}^n \rightarrow \mathbb{R}^m$. Then $\frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}}$ is a Jacobian matrix. However, the backpropagation algorithm is analogous.

Hidden Layers Derivatives

- σ:

$$\frac{\partial\sigma(x)}{\partial x} = \sigma(x)\cdot\big(1-\sigma(x)\big)$$

- $\tanh$:

$$\frac{\partial\tanh(x)}{\partial x} = 1-\tanh(x)^2$$

- ReLU:

$$\frac{\partial\operatorname{ReLU}(x)}{\partial x} = \left\{\begin{matrix} 1 & \text{if } x > 0 \\ \text{NaN} & \text{if } x = 0 \\ 0 & \text{if } x < 0 \end{matrix}\right\} \xlongequal{\text{assuming } \frac{\partial\operatorname{ReLU}(x)}{\partial x}(0)=0} \big[x > 0\big] = \big[\operatorname{ReLU}(x) > 0\big]$$

Stochastic Gradient Descents

Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) Algorithm

Input: NN computing function $f(\boldsymbol{x}; \boldsymbol{\theta})$ with initial value of parameters $\boldsymbol{\theta}$.

Input: Learning rate α.

Output: Updated parameters $\boldsymbol{\theta}$.

- Repeat until stopping criterion is met:
  - Sample a minibatch of m training examples $(\boldsymbol{x}^{(i)}, y^{(i)})$
    - in theory, we could sample each minibatch independently;
    - however, almost every time we want to process all training instances before repeating them, which can be implemented by generating a random permutation and then splitting it into minibatch-sized chunks
      - one pass through the data is called an **epoch**
  - $\boldsymbol{g} \leftarrow \frac{1}{m} \sum_i \nabla_{\boldsymbol{\theta}} L\big(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), y^{(i)}\big)$
  - $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \boldsymbol{g}$

SGD With Momentum

SGD With Momentum

Input: NN computing function $f(\boldsymbol{x};\boldsymbol{\theta})$ with initial value of parameters $\boldsymbol{\theta}$.

Input: Learning rate α, momentum β.

Output: Updated parameters $\boldsymbol{\theta}$.

- $\boldsymbol{v} \leftarrow \mathbf{0}$
- Repeat until stopping criterion is met:
  - Sample a minibatch of m training examples $(\boldsymbol{x}^{(i)}, y^{(i)})$
  - $\boldsymbol{g} \leftarrow \frac{1}{m}\sum_i \nabla_{\boldsymbol{\theta}} L\big(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), y^{(i)}\big)$
  - $\boldsymbol{v} \leftarrow \beta\boldsymbol{v} + \boldsymbol{g}$
  - $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha\boldsymbol{v}$

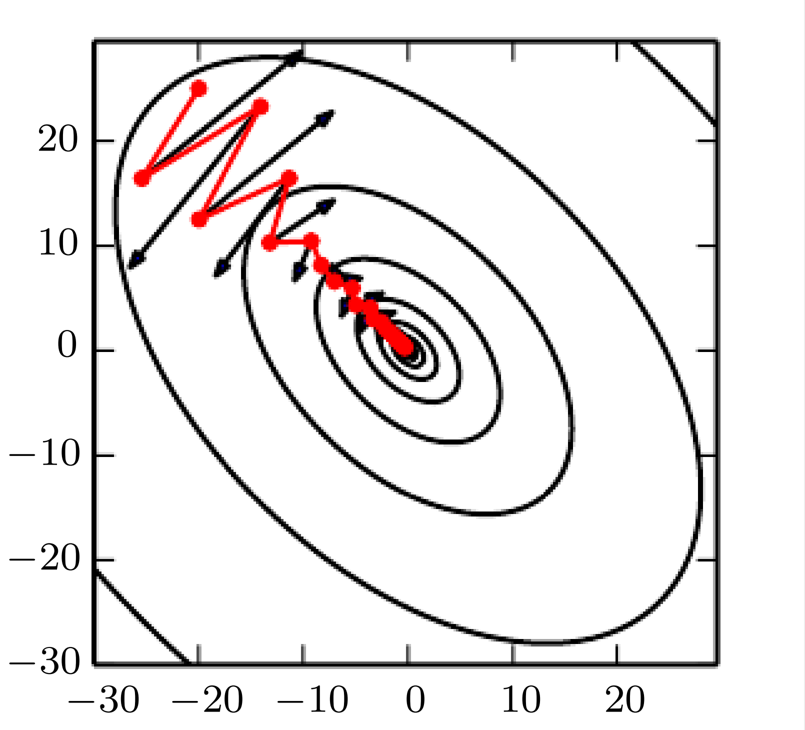


Figure 8.5 of "Deep Learning" book, https://www.deeplearningbook.org

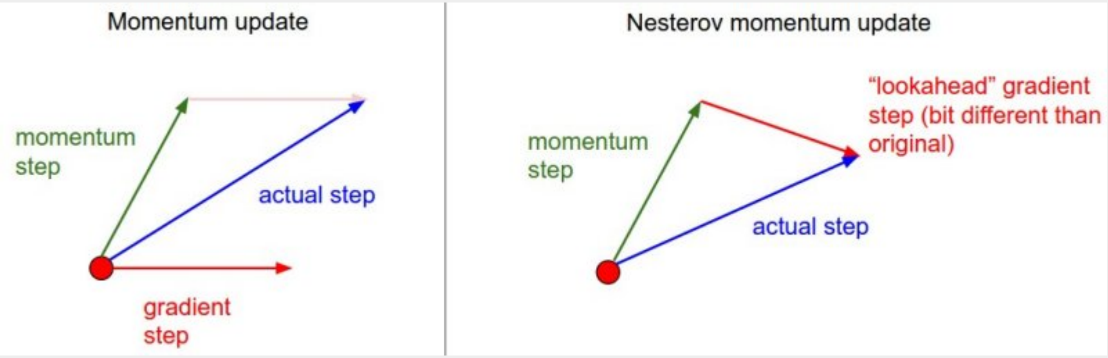
A nice writeup about momentum can be found on https://distill.pub/2017/momentum/.

SGD With Nesterov Momentum

SGD With Nesterov Momentum

```
Input: NN computing function $f(\boldsymbol{x}; \boldsymbol{\theta})$ with initial value of parameters $\boldsymbol{\theta}$.
Input: Learning rate $\alpha$, momentum $\beta$.
Output: Updated parameters $\boldsymbol{\theta}$.

• $\boldsymbol{v} \leftarrow \mathbf{0}$
• Repeat until stopping criterion is met:
    ○ Sample a minibatch of $m$ training examples $(\boldsymbol{x}^{(i)}, y^{(i)})$
    ○ $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha\beta\boldsymbol{v}$
    ○ $\boldsymbol{g} \leftarrow \frac{1}{m}\sum_i \nabla_{\boldsymbol{\theta}} L\big(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), y^{(i)}\big)$
    ○ $\boldsymbol{v} \leftarrow \beta\boldsymbol{v} + \boldsymbol{g}$
    ○ $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha\boldsymbol{g}$        without Nesterov $\boldsymbol{\theta} - \alpha(\beta\boldsymbol{v} + \boldsymbol{g})$
```



https://github.com/cs231n/cs231n.github.io/blob/master/assets/nn3/nesterov.jpeg

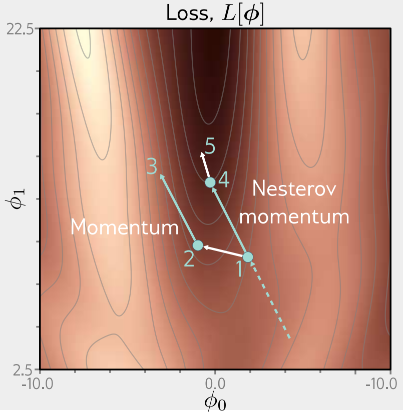


Figure 6.8 of "Understanding Deep Learning", https://udlbook.github.io/udlbook/

Algorithms with Adaptive Learning Rates

Algorithms with Adaptive Learning Rates

AdaGrad (2011)

```
**Input**: NN computing function $f(\boldsymbol{x};\boldsymbol{\theta})$ with initial value of parameters $\boldsymbol{\theta}$.
**Input**: Learning rate $\alpha$, constant $\varepsilon$ (usually $10^{-7}$).
**Output**: Updated parameters $\boldsymbol{\theta}$.

• $\boldsymbol{r} \leftarrow \mathbf{0}$
• Repeat until stopping criterion is met:
    ◦ Sample a minibatch of $m$ training examples $(\boldsymbol{x}^{(i)}, y^{(i)})$
    ◦ $\boldsymbol{g} \leftarrow \frac{1}{m}\sum_i \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), y^{(i)})$
    ◦ $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g}^2$
    ◦ $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \frac{\alpha}{\sqrt{\boldsymbol{r}+\varepsilon}}\boldsymbol{g}$
```

- The $\boldsymbol{g}^2$ and $\frac{\alpha}{\sqrt{\boldsymbol{r}+\varepsilon}}\boldsymbol{g}$ are computed element-wise, i.e., $\boldsymbol{g}^2 = \boldsymbol{g} \odot \boldsymbol{g}$. It might be better to write $\frac{\alpha}{\sqrt{\boldsymbol{r}+\varepsilon}} \odot \boldsymbol{g}$, but it is not done in the papers, so we are keeping the usual notation.

Algorithms with Adaptive Learning Rates

AdaGrad has favourable convergence properties (being faster than regular SGD) for convex loss landscapes. In this settings, gradients converge to zero reasonably fast.

However, for nonconvex losses, gradients can stay quite large for a long time. In that case, the algorithm behaves as if decreasing learning rate by a factor of $1/\sqrt{t}$, because if each

$$\boldsymbol{g} \approx \boldsymbol{g}_0,$$

then after t steps

$$\boldsymbol{r} \approx t \cdot \boldsymbol{g}_0^2,$$

and therefore

$$\frac{\alpha}{\sqrt{\boldsymbol{r}} + \varepsilon} \approx \frac{\alpha/\sqrt{t}}{\sqrt{\boldsymbol{g}_0^2} + \varepsilon/\sqrt{t}}.$$

Algorithms with Adaptive Learning Rates

RMSProp (2012)

Input: NN computing function $f(\boldsymbol{x};\boldsymbol{\theta})$ with initial value of parameters $\boldsymbol{\theta}$.

Input: Learning rate α, momentum β (usually 0.9), constant ε (usually 10^{-7}).

Output: Updated parameters $\boldsymbol{\theta}$.

```
• $\boldsymbol{r} \leftarrow \boldsymbol{0}$
• Repeat until stopping criterion is met:
    ○ Sample a minibatch of $m$ training examples $(\boldsymbol{x}^{(i)}, y^{(i)})$
    ○ $\boldsymbol{g} \leftarrow \frac{1}{m}\sum_i \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), y^{(i)})$
    ○ $\boldsymbol{r} \leftarrow \beta\boldsymbol{r} + (1-\beta)\boldsymbol{g}^2$
    ○ $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \frac{\alpha}{\sqrt{\boldsymbol{r}+\varepsilon}}\boldsymbol{g}$
```

However, after first step, $\boldsymbol{r} = (1-\beta)\boldsymbol{g}^2$, which for default $\beta = 0.9$ is

$$\boldsymbol{r} = 0.1\boldsymbol{g}^2,$$

so $\boldsymbol{r}$ is a biased estimate of $\mathbb{E}[\boldsymbol{g}^2]$ (but the bias converges to zero exponentially fast).

Algorithms with Adaptive Learning Rates

Adam (2014)

```
Input: NN computing function $f(\boldsymbol{x};\boldsymbol{\theta})$ with initial value of parameters $\boldsymbol{\theta}$.
Input: Learning rate $\alpha$ (default 0.001), constant $\varepsilon$ (usually $10^{-7}$).
Input: Momentum $\beta_1$ (default 0.9), momentum $\beta_2$ (default 0.999).
Output: Updated parameters $\boldsymbol{\theta}$.

• $\boldsymbol{s} \leftarrow \mathbf{0}$, $\boldsymbol{r} \leftarrow \mathbf{0}$, $t \leftarrow 0$
• Repeat until stopping criterion is met:
    ○ Sample a minibatch of $m$ training examples $(\boldsymbol{x}^{(i)}, y^{(i)})$
    ○ $\boldsymbol{g} \leftarrow \frac{1}{m}\sum_i \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), y^{(i)})$
    ○ $t \leftarrow t + 1$
    ○ $\boldsymbol{s} \leftarrow \beta_1 \boldsymbol{s} + (1-\beta_1)\boldsymbol{g}$                    (biased first moment estimate)
    ○ $\boldsymbol{r} \leftarrow \beta_2 \boldsymbol{r} + (1-\beta_2)\boldsymbol{g}^2$                  (biased second moment estimate)
    ○ $\hat{\boldsymbol{s}} \leftarrow \boldsymbol{s}/(1-\beta_1^t)$, $\hat{\boldsymbol{r}} \leftarrow \boldsymbol{r}/(1-\beta_2^t)$    (unbiased estimates of the moments)
    ○ $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \frac{\alpha}{\sqrt{\hat{\boldsymbol{r}}}+\varepsilon}\hat{\boldsymbol{s}}$
```

Adam Bias Correction

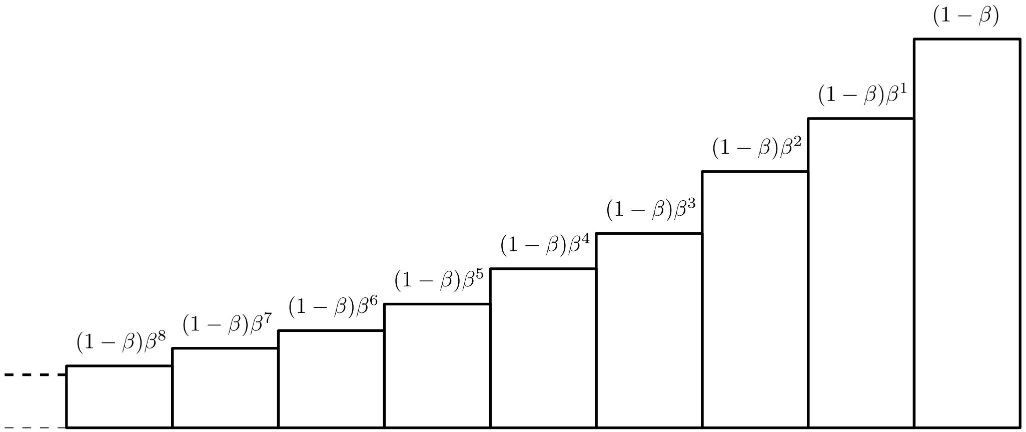
To allow analysis, we add indices to the update

$$\boldsymbol{s}_t \leftarrow \beta_1 \boldsymbol{s}_{t-1} + (1-\beta_1)\boldsymbol{g}_t,$$

with $\boldsymbol{s}_0 \leftarrow \mathbf{0}$.

After t steps, we have

$$\boldsymbol{s}_t = (1-\beta_1)\sum_{i=1}^{t} \beta_1^{t-i}\boldsymbol{g}_i.$$



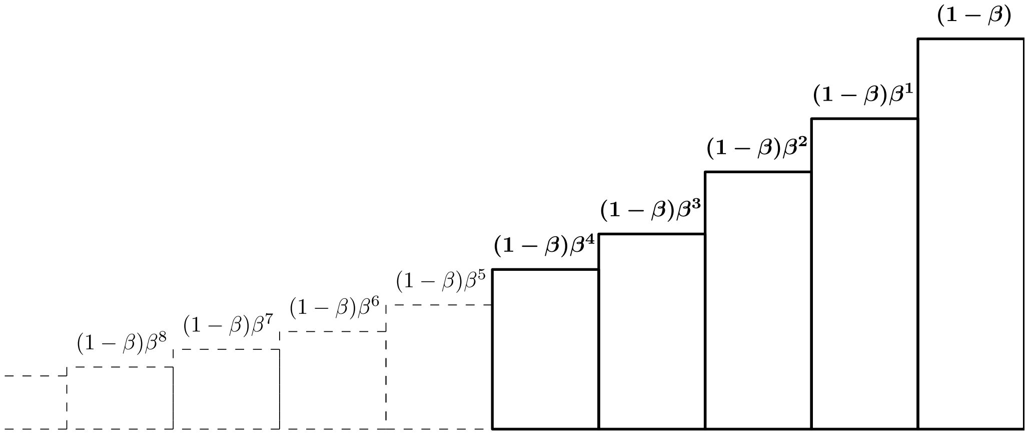
Because $\sum_{i=0}^{\infty}\beta_1^i = \frac{1}{1-\beta_1}$, $\boldsymbol{s}_\infty$ is computed as a weighted average of infinitely many elements.

Adam Bias Correction

However, for $t < \infty$, the sum of weights in the computation of $\boldsymbol{s}_t$ does not sum to one.

To obtain an unbiased estimate, we therefore need to account for the "missing" elements; in other words, we need to scale the weights so that they sum to one.

The sum of weights after t steps is



$$(1-\beta_1)\sum_{i=1}^{t}\beta_1^{t-i} = \sum_{i=1}^{t}\beta_1^{t-i} - \sum_{i=0}^{t-1}\beta_1^{t-i} = 1-\beta_1^t,$$

so we obtain an unbiased estimate by dividing $\boldsymbol{s}_t$ with $(1-\beta_1^t)$, and analogously for the correction of $\boldsymbol{r}$.

Visualization of Gradient Normalization Effect

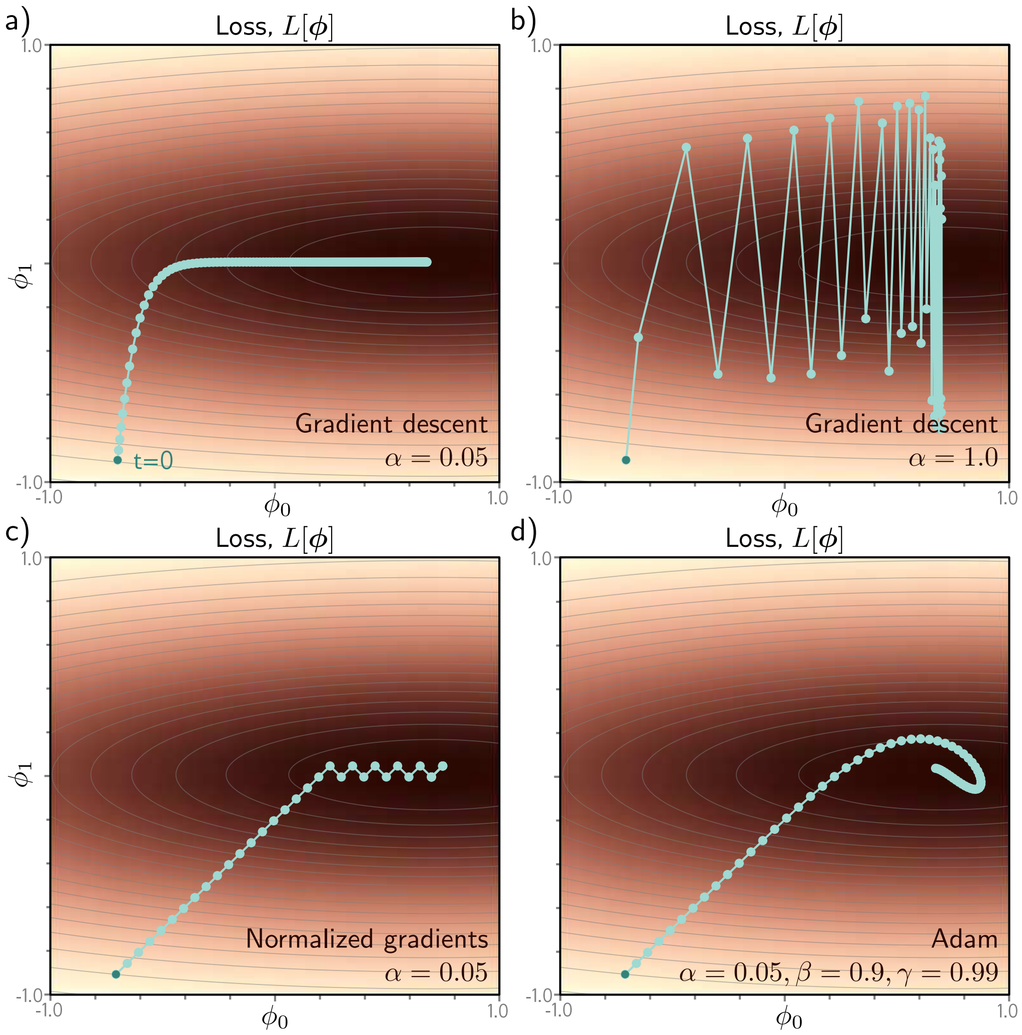
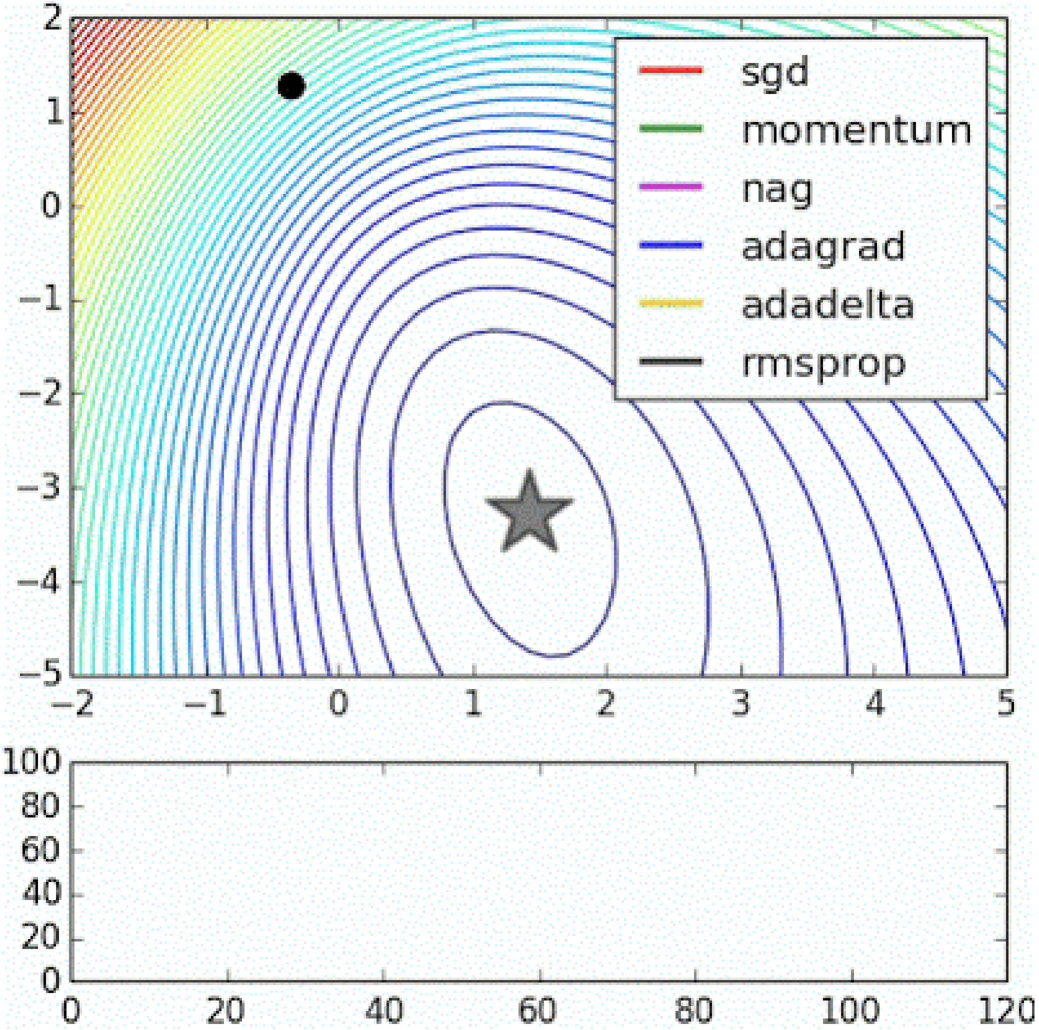


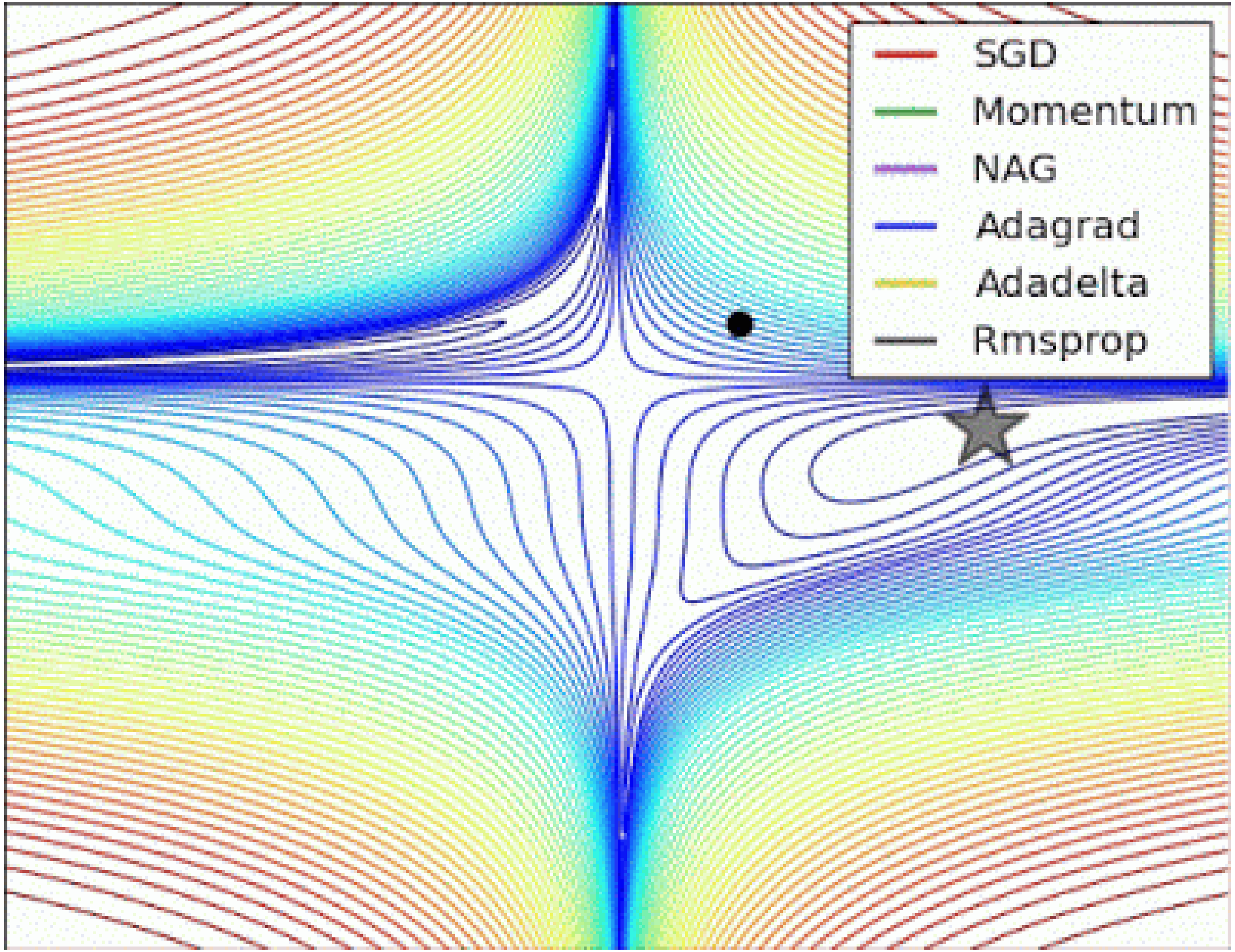
Figure 6.9 of "Understanding Deep Learning", https://udlbook.github.io/udlbook/

Adaptive Optimizers Animations



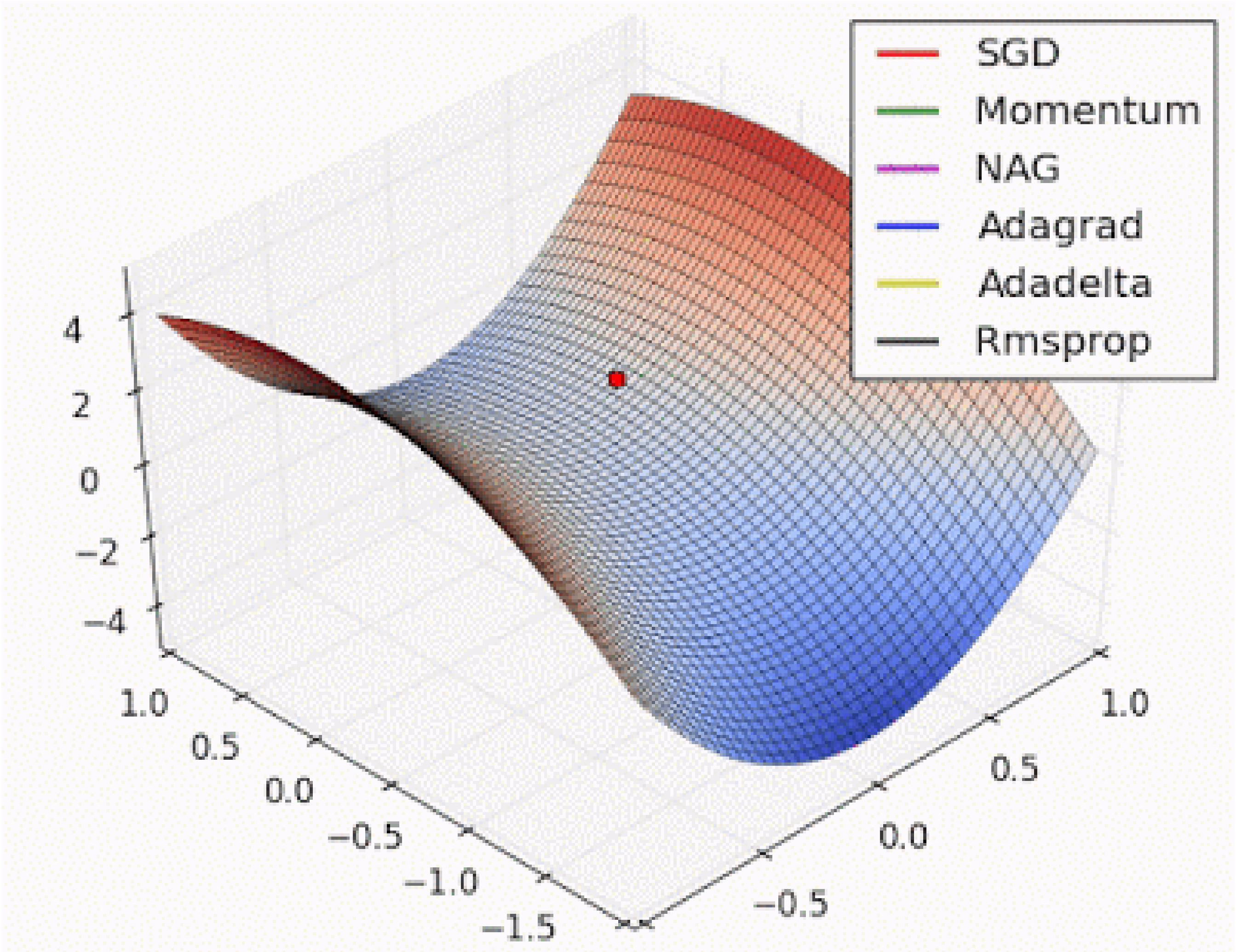
http://2.bp.blogspot.com/-q6l20Vs4P_w/VPmIC7sEhnI/AAAAAAAACC4/g3UOUX2r_yA/s400/ found at http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html

Adaptive Optimizers Animations



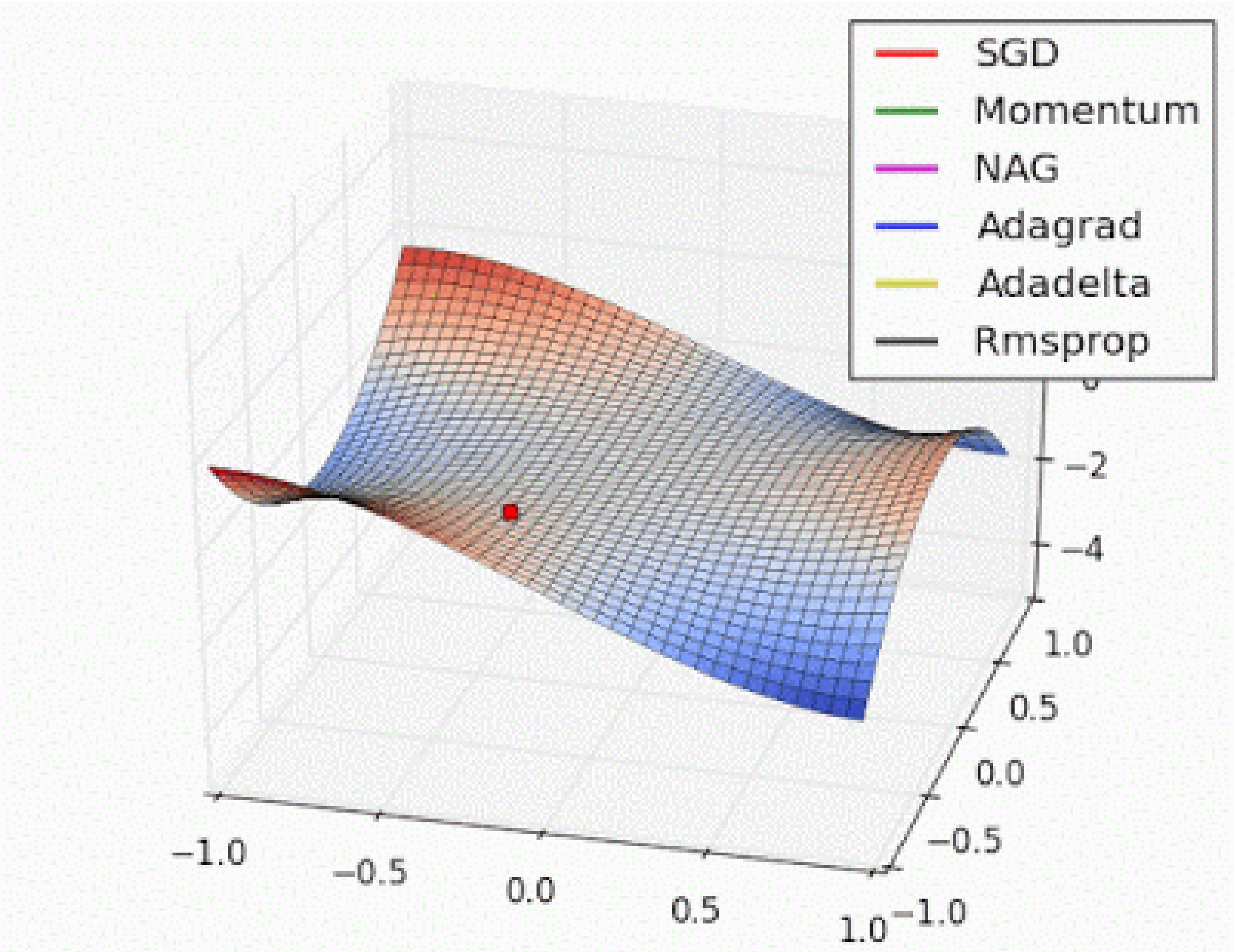
http://2.bp.blogspot.com/-L98w-SBmF58/VPmICIjKEKI/AAAAAAAACCs/rrFz3VetYmM/s400/ found at http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html

Adaptive Optimizers Animations



http://3.bp.blogspot.com/-nrtJPrdBWuE/VPmIB46F2aI/AAAAAAAACCw/vaE_B0SVy5k/s400/ found at http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html

Adaptive Optimizers Animations



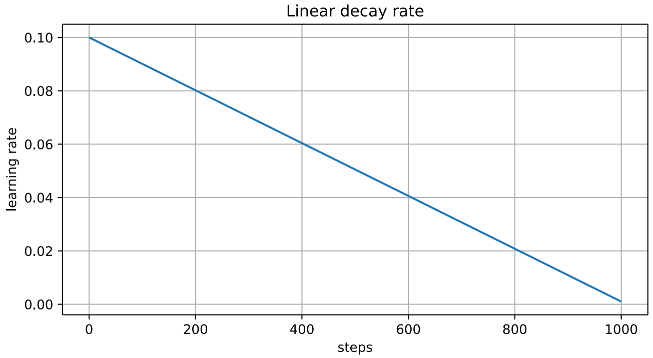
http://1.bp.blogspot.com/-K_X-yud8nj8/VPmIBxwGlsI/AAAAAAAACC0/JS-h1fa09EQ/s400/ found at http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html

Learning Rate Schedules

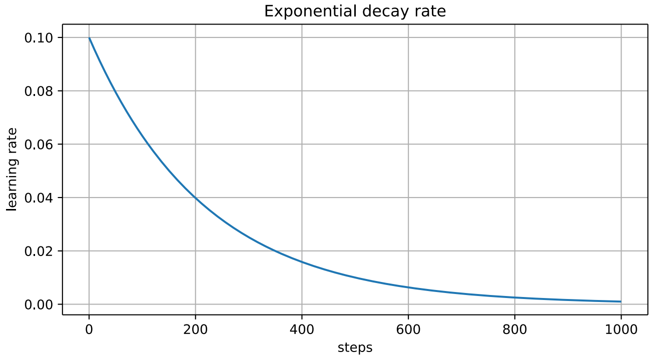
Learning Rate Schedules

Even if RMSProp and Adam are adaptive, they still usually require carefully tuned decreasing learning rate for top-notch performance.

- **Polynomial decay**: learning rate is multiplied by some polynomial of the current update number t.
  - **Linear decay** uses $\alpha_t = \alpha_\textrm{initial} \cdot \left(1 - \frac{t}{\textrm{max steps}}\right)$ and has theoretical guarantees of convergence, but is usually too fast for deep neural networks.
  - **Inverse square root decay** uses $\alpha_t = \alpha_\textrm{initial} \cdot \frac{1}{\sqrt{t}}$ and is currently used by best machine translation models.



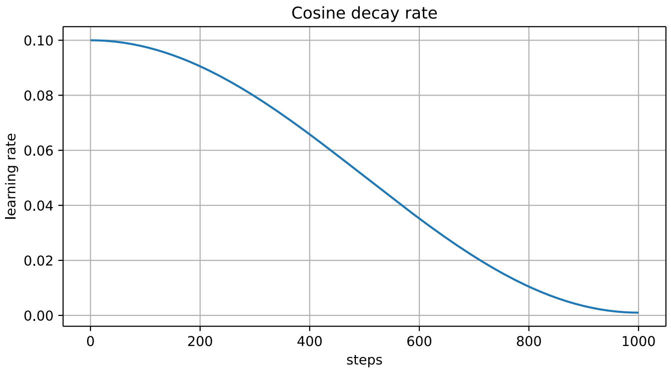
- **Exponential decay**: learning rate is multiplied by a constant each minibatch/epoch/several epochs.
  - $\alpha_t = \alpha_\textrm{initial} \cdot c^t$
  - Often used for convolutional networks (image recognition etc.).



Learning Rate Schedules

- **Cosine decay**: The cosine decay has became quite popular in the past years, both for training and finetuning.

$$\begin{aligned}\alpha_t &= \alpha_\text{initial} \cdot \frac{1}{2}\left(1 + \cos\left(\pi \cdot \frac{t}{\text{max steps}}\right)\right)\\ &= \alpha_\text{initial} \cdot \cos^2\left(\frac{\pi}{2} \cdot \frac{t}{\text{max steps}}\right)\end{aligned}$$



- Cyclic restarts, warmup, ...

The `torch.optim.lr_scheduler` offers several such learning rate schedules.

- `torch.optim.lr_scheduler.LinearLR`, `torch.optim.lr_scheduler.PolynomialLR`
- `torch.optim.lr_scheduler.ExponentialLR`
- `torch.optim.lr_scheduler.StepLR`, `torch.optim.lr_scheduler.MultiStepLR`
- `torch.optim.lr_scheduler.CosineAnnealingLR`
- `torch.optim.lr_scheduler.LambdaLR`

Summary

- We train neural networks by minimizing **training error** on the training data.
- We measure the performance of a model by evaluating **generalization error** on unseen examples (usually a fixed test set).

The training process is based on several components:

- **Maximum likelihood estimation** provides a loss for any output distribution. This loss is in some sense the most statistically efficient.
- **Gradient descent** is an algorithm for performing the training itself, based on the (stochastic) gradient of the loss function.
- **Backpropagation** is an efficient algorithm for computing the gradient of the loss function in a neural network.
- **Stochastic** gradient descent with minibatches allows efficient training even for very large training sets.
- **Adam** is an improved variant of SGD and is the algorithm of choice for training most neural networks. It scales the learning rate adaptively for every parameter.

Extra Content

Why do NNs Generalize so Well: Double Descent

Why do Neural Networks Generalize so Well: Double Descent

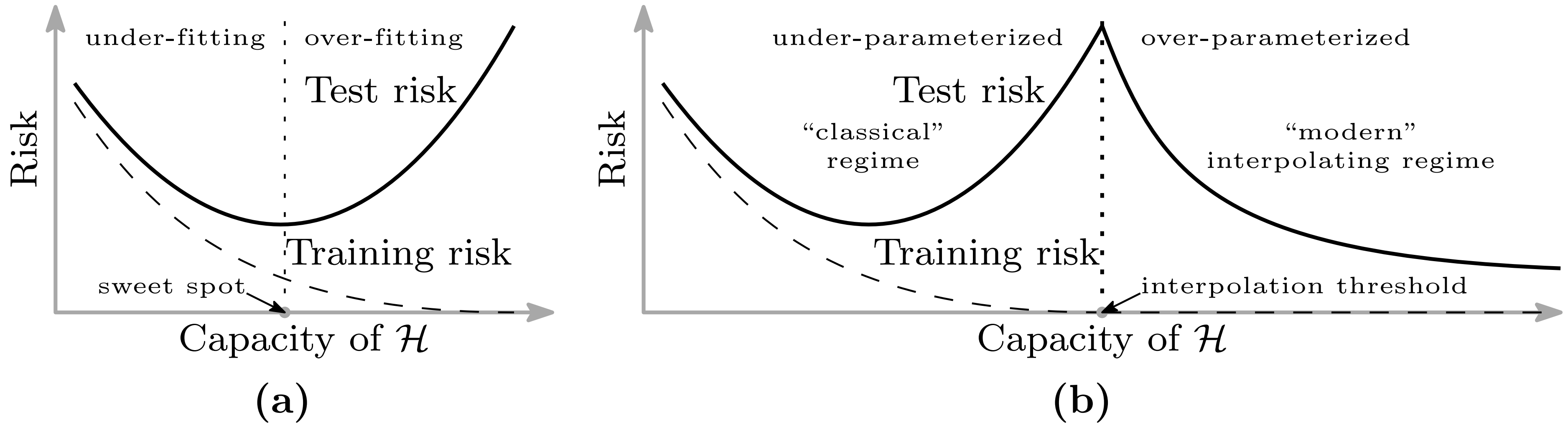


Figure 1: **Curves for training risk (dashed line) and test risk (solid line).** (**a**) The classical *U-shaped risk curve* arising from the bias-variance trade-off. (**b**) The *double descent risk curve*, which incorporates the U-shaped risk curve (i.e., the "classical" regime) together with the observed behavior from using high capacity function classes (i.e., the "modern" interpolating regime), separated by the interpolation threshold. The predictors to the right of the interpolation threshold have zero training risk.

Figure 1 of "Reconciling modern machine learning practice and the bias-variance trade-off", https://arxiv.org/abs/1812.11118

Deep Double Descent

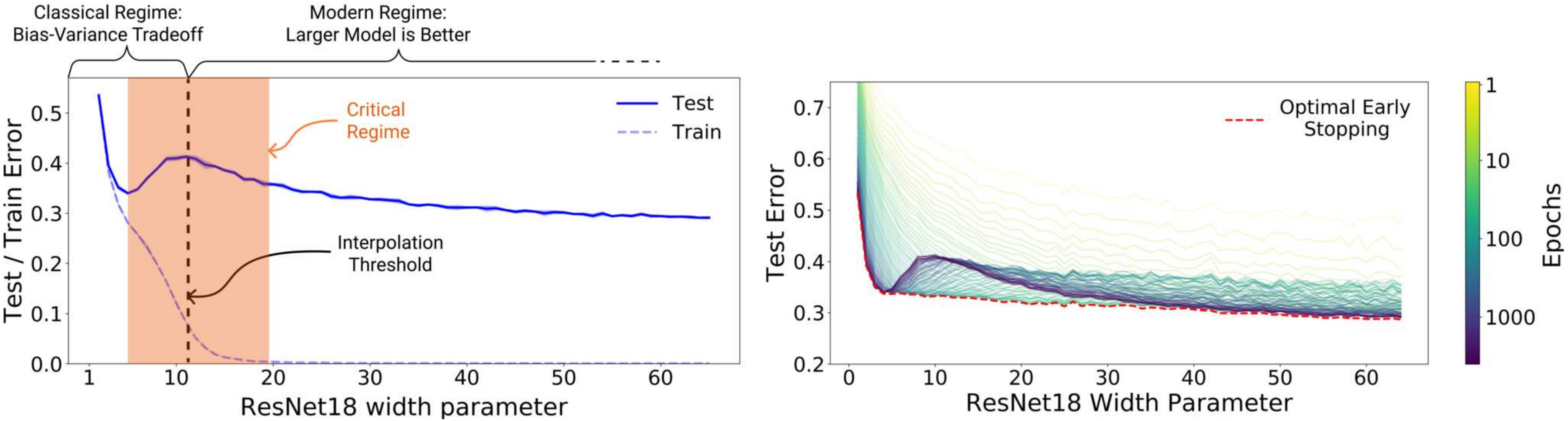


Figure 1: **Left:** Train and test error as a function of model size, for ResNet18s of varying width on CIFAR-10 with 15% label noise. **Right:** Test error, shown for varying train epochs. All models trained using Adam for 4K epochs. The largest model (width 64) corresponds to standard ResNet18.

Figure 1 of "Deep Double Descent: Where Bigger Models and More Data Hurt", https://arxiv.org/abs/1912.02292

Deep Double Descent – Effective Model Complexity

The authors define the **Effective Model Complexity** (**EMC**) of a training procedure $\mathcal{T}$ with respect to distribution $\mathcal{D}$ and parameter $\varepsilon > 0$ as

$$\mathrm{EMC}_{\mathcal{D},\varepsilon}(\mathcal{T}) \overset{\text{def}}{=} \max\left\{n \,\middle|\, \mathbb{E}_{S\sim\mathcal{D}^n}[\mathrm{Error}_S(\mathcal{T}(S))] \leq \varepsilon\right\},$$

where $\mathrm{Error}_S(M)$ is the mean error of a model M on the train samples S.

Hypothesis: For any natural data distribution $\mathcal{D}$, neural-network-based training procedure $\mathcal{T}$, and small $\varepsilon > 0$, if we consider the task of predicting labels based on n samples from $\mathcal{D}$, then:

- **Under-parametrized regime.** If $\mathrm{EMC}_{\mathcal{D},\varepsilon}(\mathcal{T})$ is sufficiently smaller than n, any perturbation of $\mathcal{T}$ that increases its effective complexity will decrease the test error.
- **Over-parametrized regime.** If $\mathrm{EMC}_{\mathcal{D},\varepsilon}(\mathcal{T})$ is sufficiently larger than n, any perturbation of $\mathcal{T}$ that increases its effective complexity will decrease the test error.
- **Critically parametrized regime.** If $\mathrm{EMC}_{\mathcal{D},\varepsilon}(\mathcal{T}) \approx n$, then a perturbation of $\mathcal{T}$ that increases its effective complexity might decrease **or increase** the test error.

Why do Neural Networks Generalize so Well: Double Descent

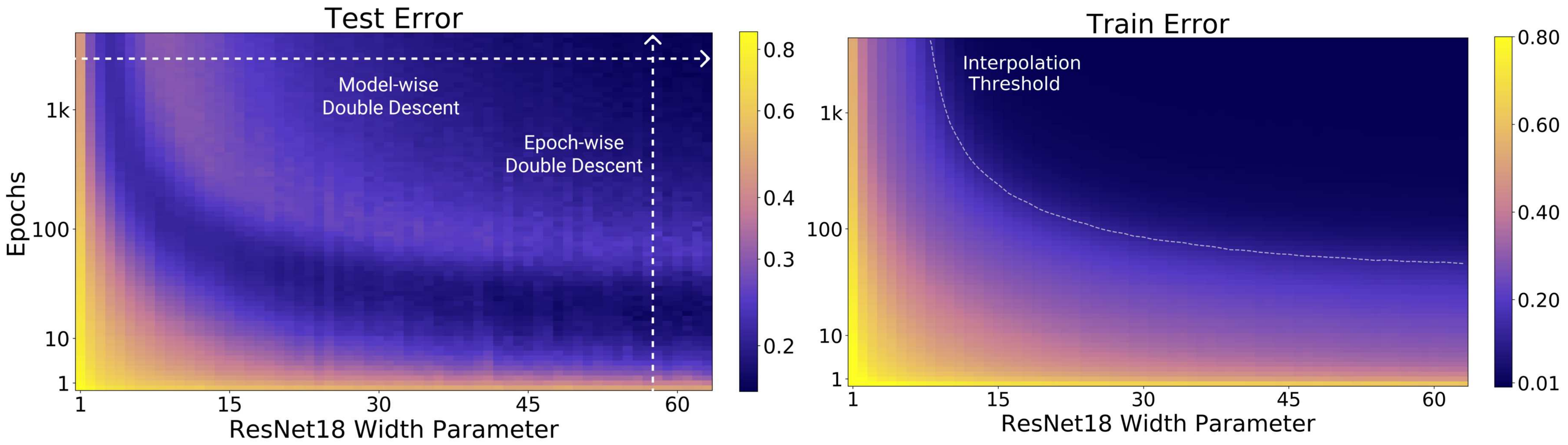
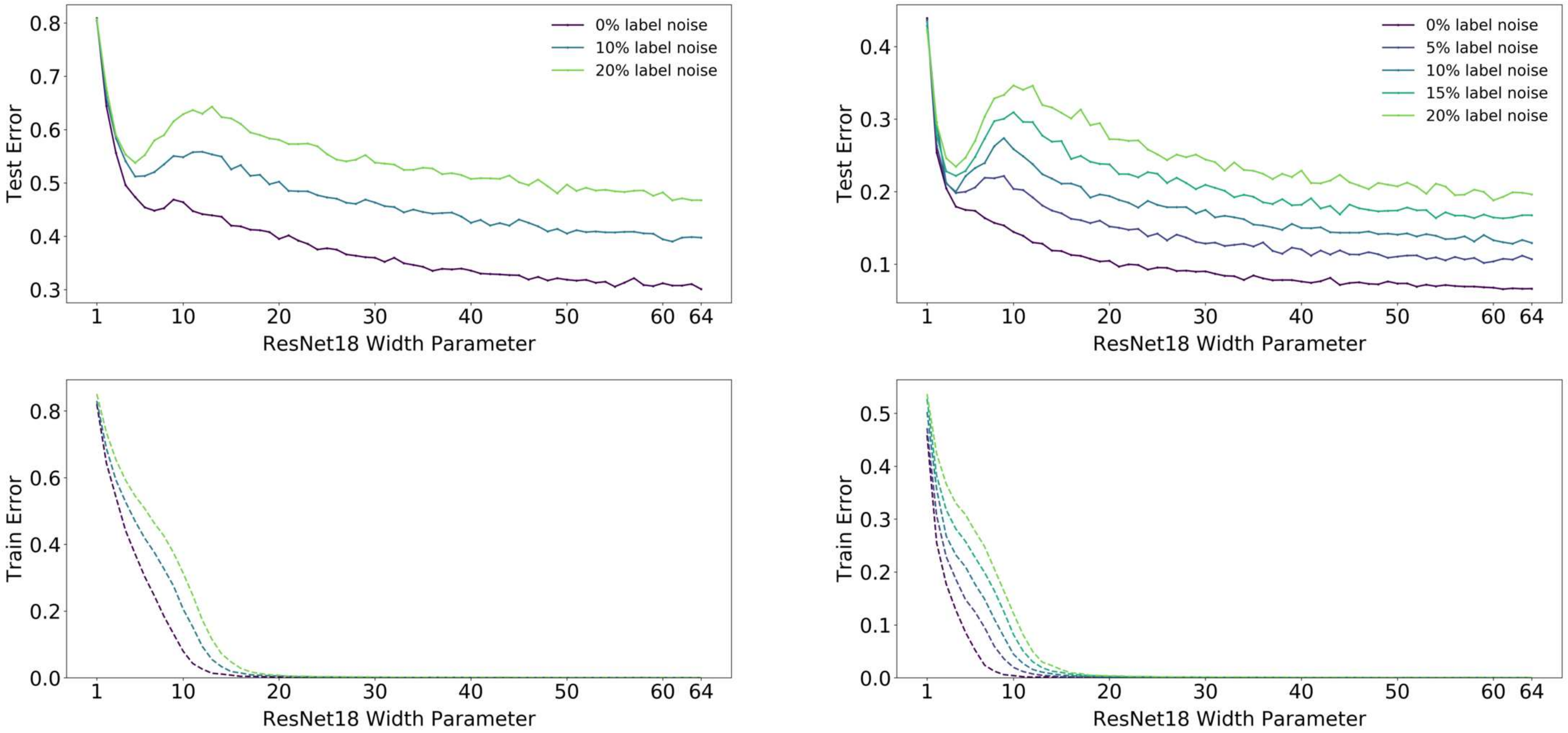


Figure 2: **Left:** Test error as a function of model size and train epochs. The horizontal line corresponds to model-wise double descent–varying model size while training for as long as possible. The vertical line corresponds to epoch-wise double descent, with test error undergoing double-descent as train time increases. **Right** Train error of the corresponding models. All models are Resnet18s trained on CIFAR-10 with 15% label noise, data-augmentation, and Adam for up to 4K epochs.

Figure 2 of "Deep Double Descent: Where Bigger Models and More Data Hurt", https://arxiv.org/abs/1912.02292

Why do Neural Networks Generalize so Well: Double Descent



(a) **CIFAR-100.** There is a peak in test error even with no label noise.

(b) **CIFAR-10.** There is a "plateau" in test error around the interpolation point with no label noise, which develops into a peak for added label noise.

Figure 4 of "Deep Double Descent: Where Bigger Models and More Data Hurt", https://arxiv.org/abs/1912.02292