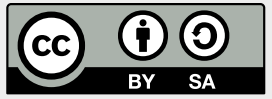
Training Neural Networks

Milan Straka

February 21, 2022

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.
  - The input tensor has usually a fixed size.
  - The input tensor is usually a vector, but it can be 2D/3D/4D
    - images, video, time sequences like speech, ...
  - The output usually describes a distribution
    - normal 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

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.

Challenges in machine learning:

- *underfitting*
- *overfitting*

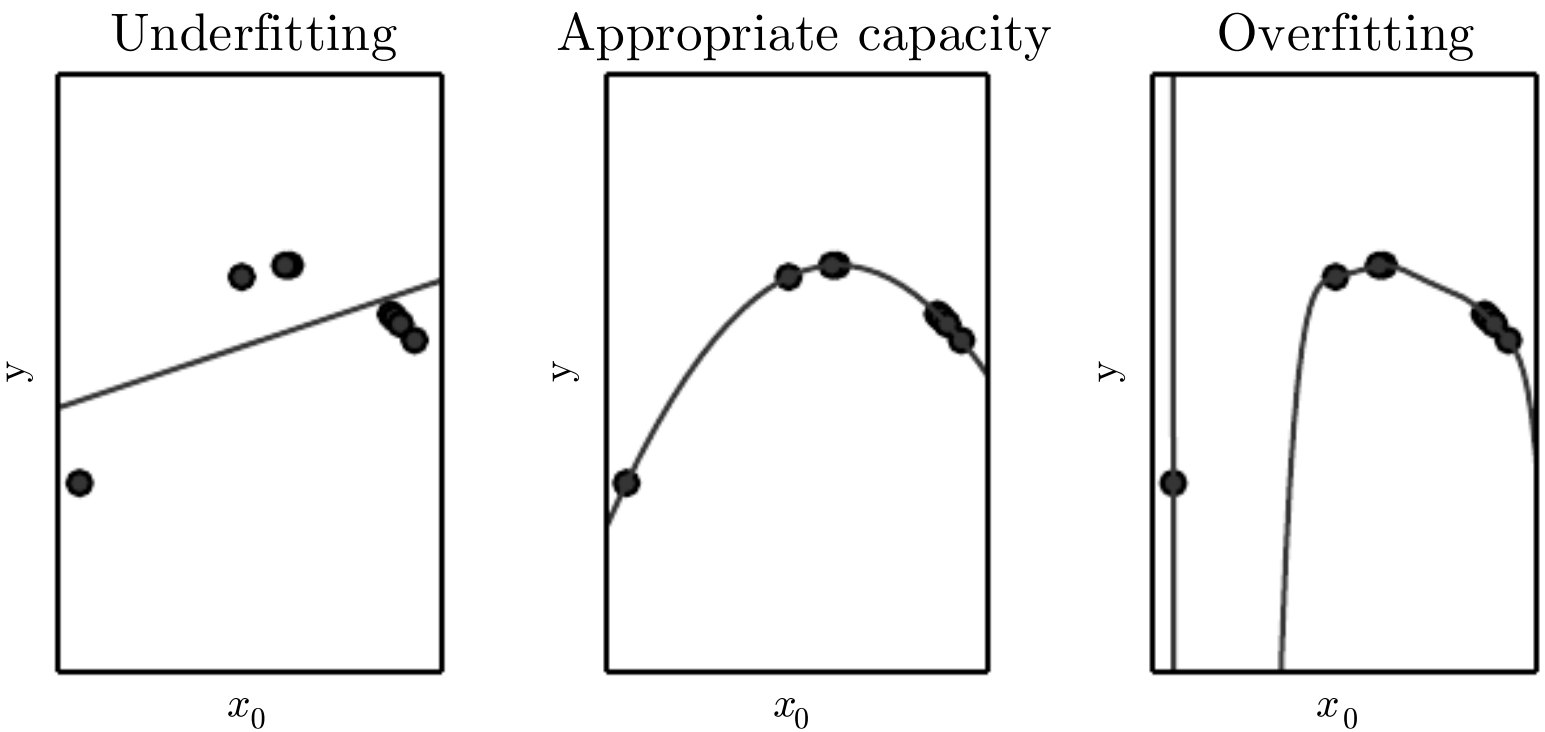


Figure 5.2 of "Deep Learning" book, https://www.deeplearningbook.org

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

- representational capacity
- effective capacity

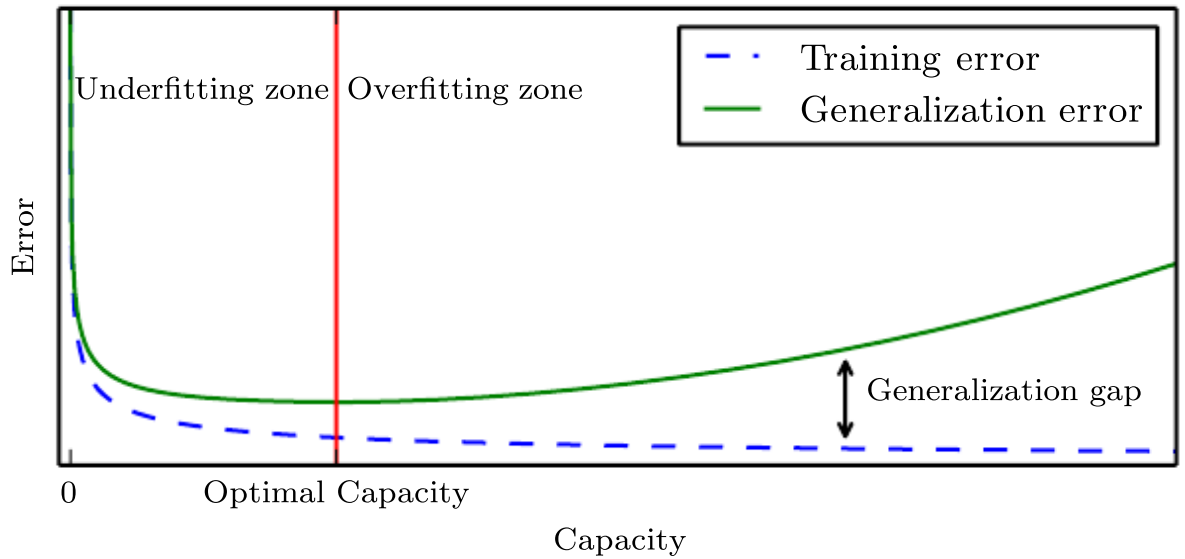


Figure 5.3 of "Deep Learning" book, https://www.deeplearningbook.org

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. In a sense, no machine learning algorithm is *universally* better than others.

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 (i.e., penalty of $\|\boldsymbol{\theta}\|^2$).

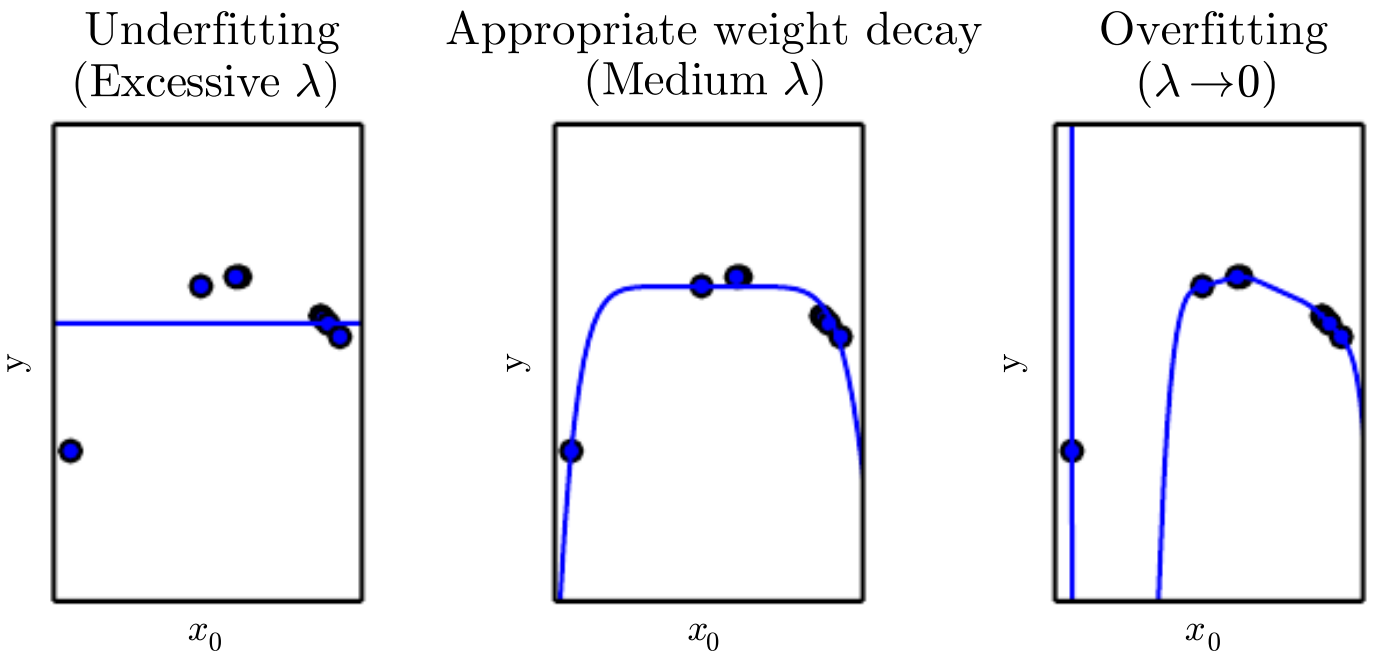


Figure 5.5 of "Deep Learning" book, https://www.deeplearningbook.org

Hyperparameters are not adapted by the learning algorithm itself.

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

Why do Neural Networks Generalize so Well

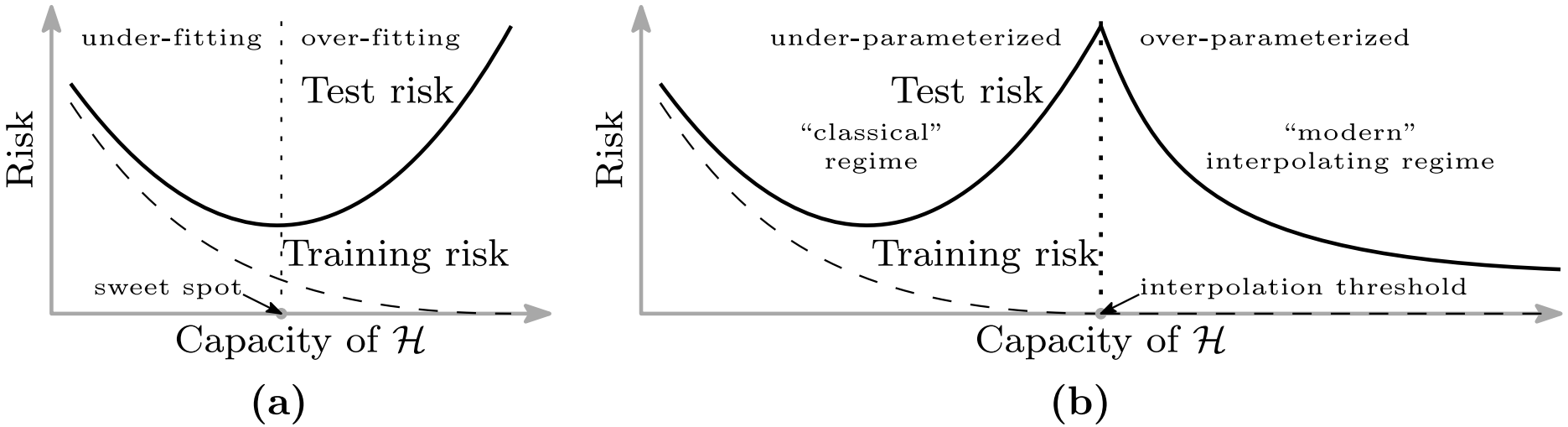


Figure 1: **Curves for training risk (dashed line) and test risk (solid line).** (**a**) The classical *U-shaped risk curve* arising from the bias-variance trade-off. (**b**) The *double descent risk curve*, which incorporates the U-shaped risk curve (i.e., the "classical" regime) together with the observed 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

Why do Neural Networks Generalize so Well

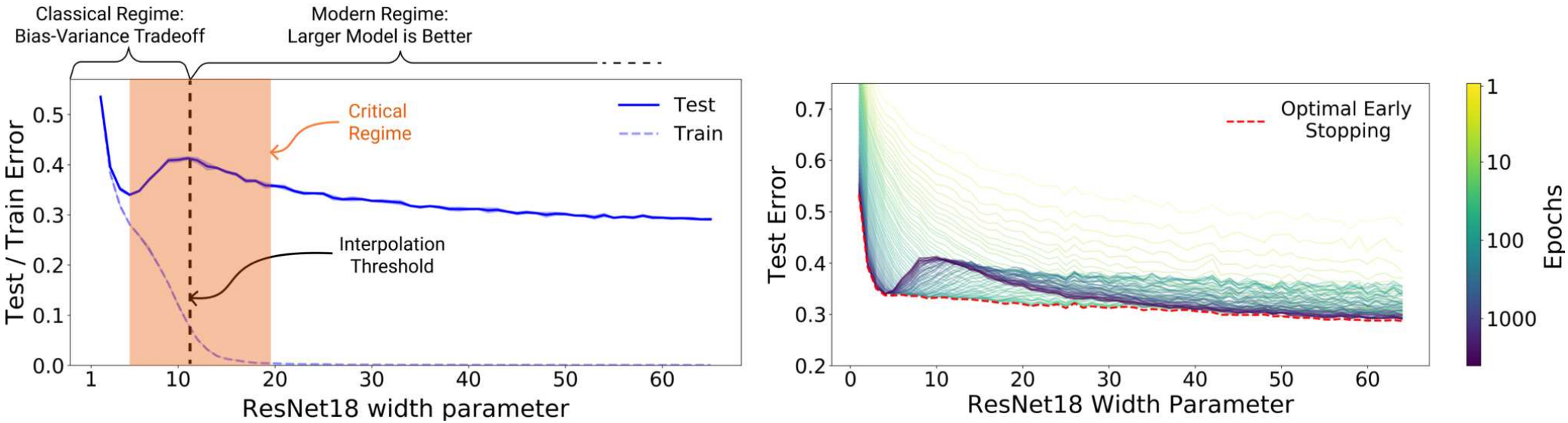
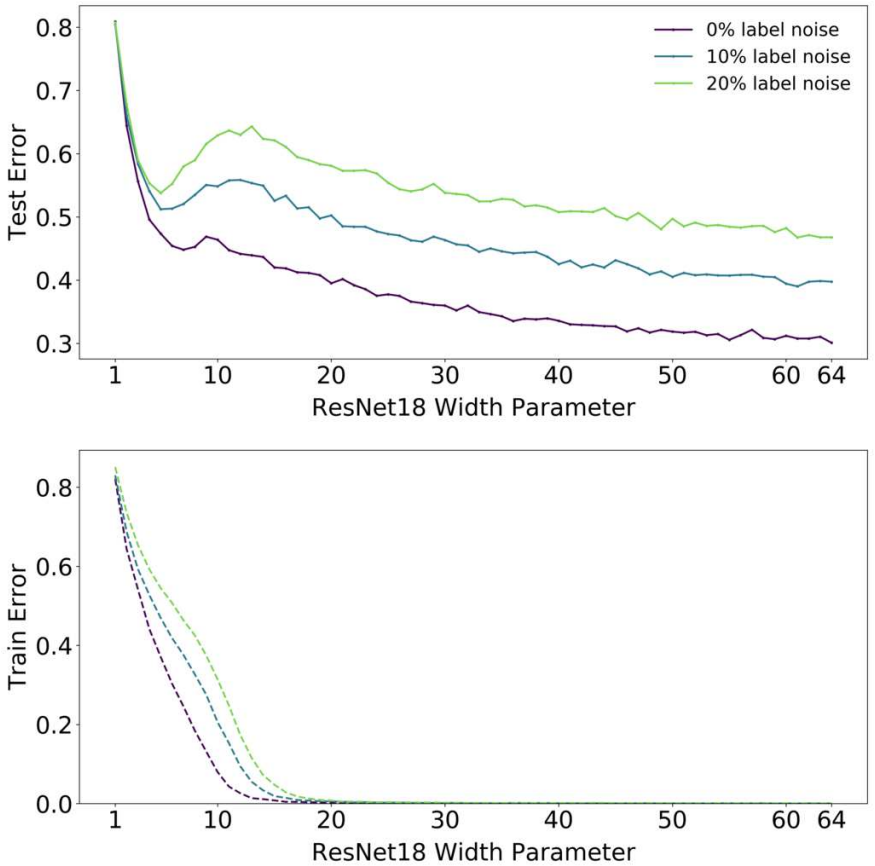


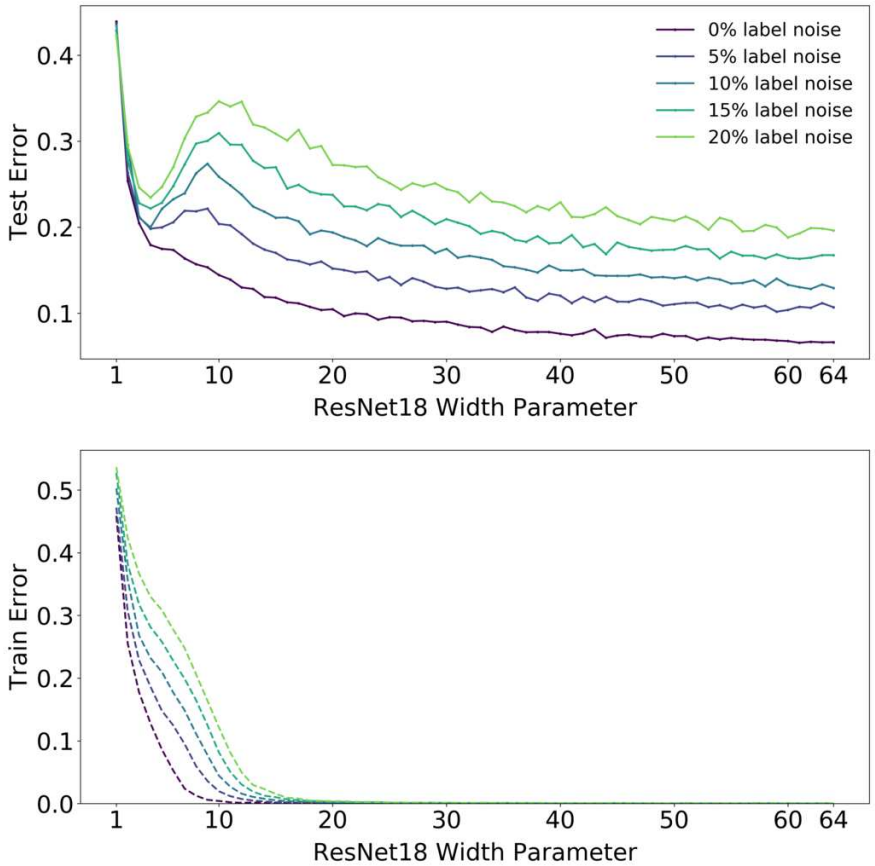
Figure 1: **Left:** Train and test error as a function of model size, for ResNet18s of varying width on CIFAR-10 with 15% label noise. **Right:** Test error, shown for varying train epochs. All models trained using Adam for 4K epochs. The largest model (width 64) corresponds to standard ResNet18.

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

Why do Neural Networks Generalize so Well



(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

Why do Neural Networks Generalize so Well

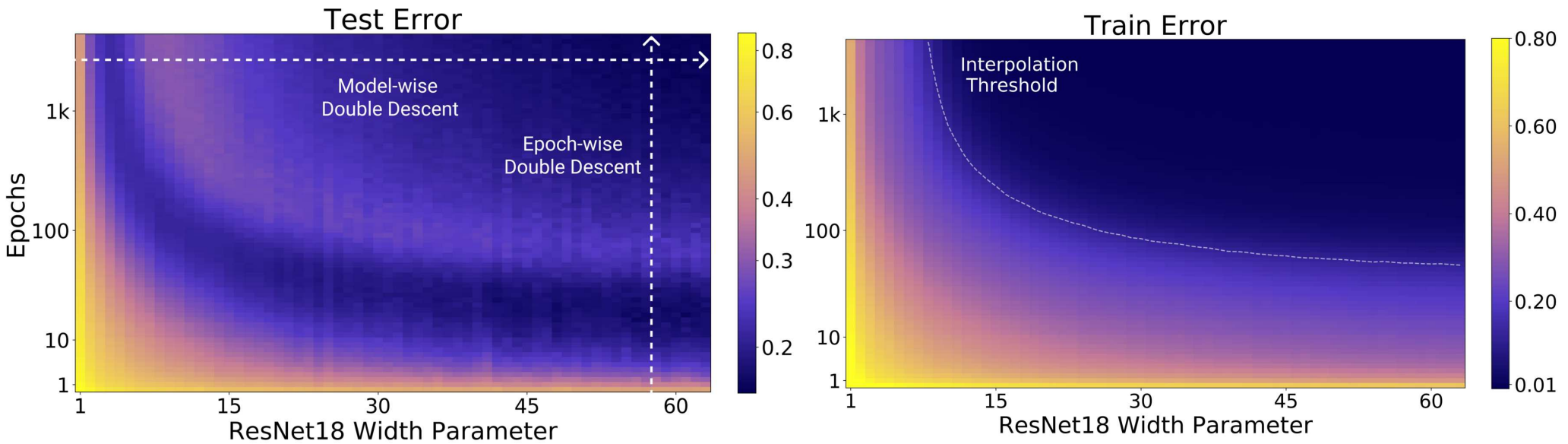


Figure 2: **Left:** Test error as a function of model size and train epochs. The horizontal line corresponds to model-wise double descent–varying model size while training for as long as possible. The vertical line corresponds to epoch-wise double descent, with test error undergoing double-descent as 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

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 $\left(\boldsymbol{x}^{(1)}, y^{(1)}\right), \left(\boldsymbol{x}^{(2)}, y^{(2)}\right), \ldots, \left(\boldsymbol{x}^{(N)}, y^{(N)}\right)$ is computed as

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

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

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

  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_{data}. We denote the empirical data distribution as $\hat{p}_{\text{data}}$ and let $p_{\text{model}}(\mathbf{x}; \boldsymbol{\theta})$ be a family of distributions.

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

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

Maximum Likelihood Estimation

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

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

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

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 we call it **biased**.

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

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

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 small variance – in some cases it can have large variance, so a biased estimator with smaller variance might be preferred.

Properties of Maximum Likelihood Estimation

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

- 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 $\varepsilon > 0$, $P(\|\boldsymbol{\theta}_m - \boldsymbol{\theta}_{p_{\text{data}}}\| > \varepsilon) \to 0$ as $m \to \infty$.

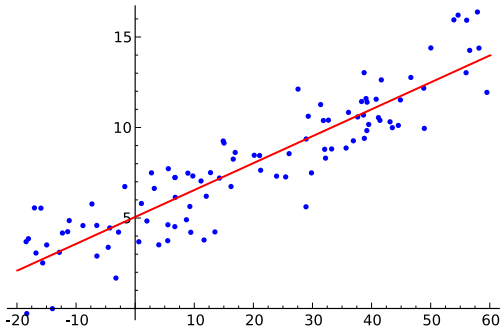
- MLE is in a sense the *most statistically efficient*. For any consistent estimator, let us consider the average distance of $\boldsymbol{\theta}_m$ and $\boldsymbol{\theta}_{p_{\text{data}}}$: $\mathbb{E}_{\mathbf{x}_1,\ldots,\mathbf{x}_m \sim p_{\text{data}}}\left[\|\boldsymbol{\theta}_m - \boldsymbol{\theta}_{p_{\text{data}}}\|^2\right]$.
  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

Assume our goal is to perform regression, i.e., to predict $p(y|\boldsymbol{x})$ for $y \in \mathbb{R}$. Let $\hat{y}(\boldsymbol{x}; \boldsymbol{\theta})$ give a prediction of the mean of y.

We define $p(y|\boldsymbol{x})$ as $\mathcal{N}(y; \hat{y}(\boldsymbol{x}; \boldsymbol{\theta}), \sigma^2)$ for a given fixed σ^2. Then:



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

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

Gradient Descent

Let a model compute $f(\boldsymbol{x}; \boldsymbol{\theta})$ using parameters $\boldsymbol{\theta}$, and for a given loss function L denote

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

Assuming we are minimizing an error function

$$\arg\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

we may use *gradient descent*:

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

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

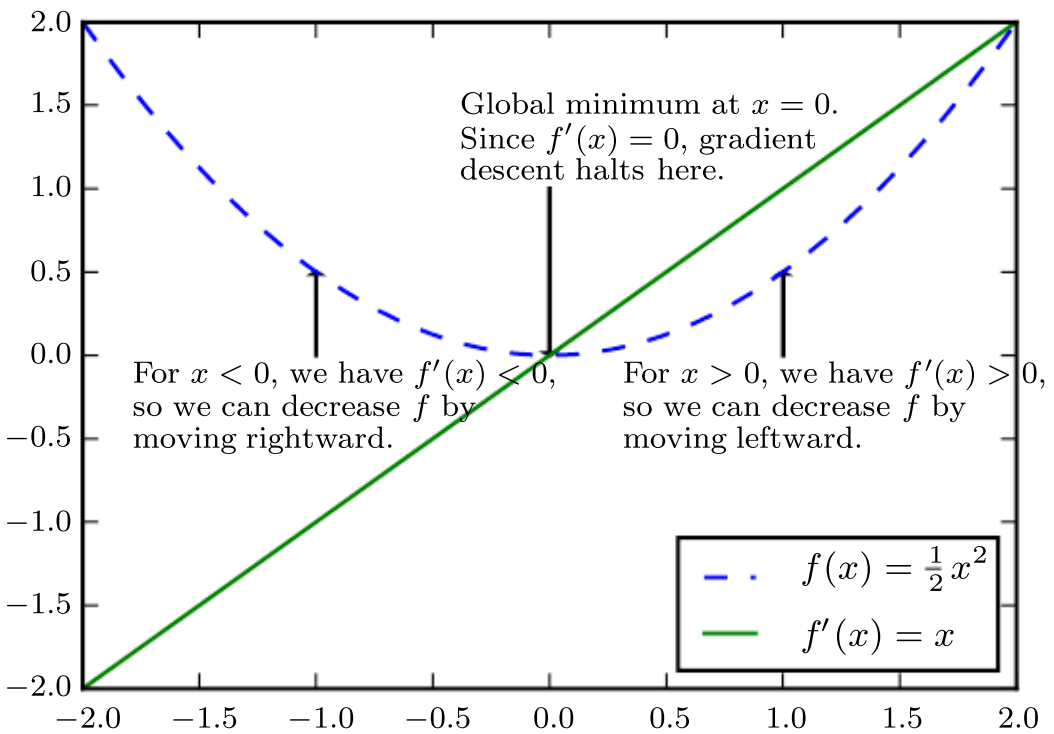


Figure 4.1 of "Deep Learning" book, https://www.deeplearningbook.org

(Standard/Batch) Gradient Descent

We use all training data to compute $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$.

Stochastic (or Online) Gradient Descent

We estimate the expectation in $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ using a single randomly sampled example from the training data. Such an estimate is unbiased, but very noisy.

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \approx \nabla_{\boldsymbol{\theta}} L\big(f(\boldsymbol{x}; \boldsymbol{\theta}), y\big) \text{ for randomly chosen } (\boldsymbol{x}, y) \text{ from } \hat{p}_{\text{data}}.$$

Minibatch SGD

The minibatch SGD is a trade-off between gradient descent and SGD – the expectation in $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ is estimated using m random independent examples from the training data.

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

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 of the real gradient $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$:

$$\boldsymbol{\theta}_{i+1} \leftarrow \boldsymbol{\theta}_i - \alpha_i \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_i).$$

It can be proven (under some reasonable conditions; see Robbins and Monro algorithm, 1951) that if the loss function is convex and continuous, then SGD converges to the unique optimum almost surely if the sequence of learning rates α_i fulfills the following conditions:

$$\forall i : \alpha_i > 0, \quad \sum_i \alpha_i = \infty, \quad \sum_i \alpha_i^2 < \infty.$$

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

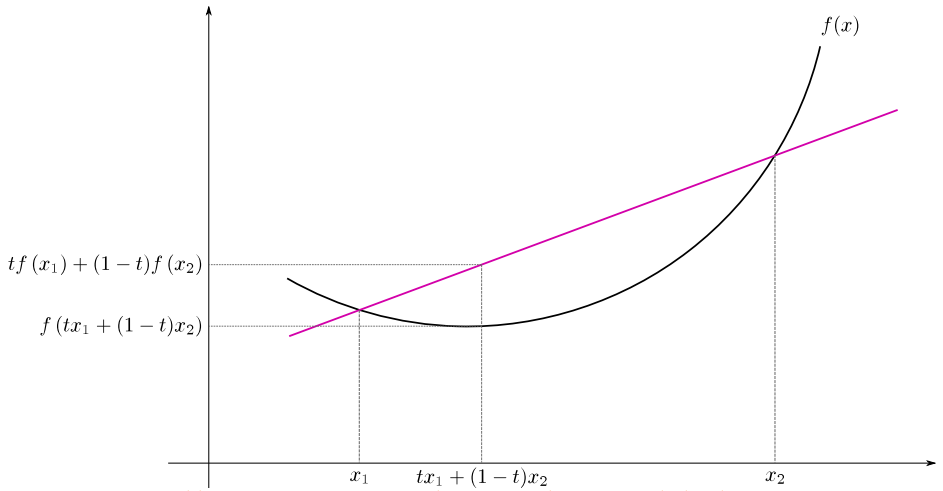
For nonconvex loss functions, we can get guarantees of converging to a *local* optimum only.

Note that finding a global minimum of an arbitrary function is *at least NP-hard*.

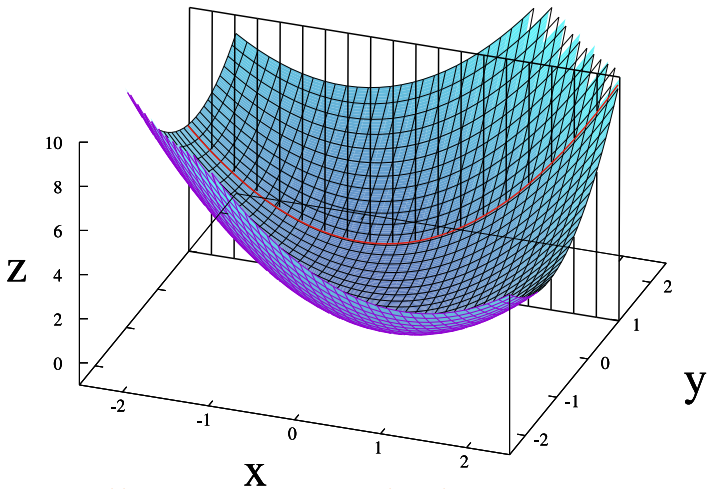
Stochastic Gradient Descent Convergence

Convex functions mentioned on the previous slide are such that for x_1, x_2 and real $0 \leq t \leq 1$,

$$f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2).$$



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



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

A twice-differentiable function is convex iff its second derivative is always nonnegative.

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

Well-known examples of convex functions are x^2, e^x and $-\log x$.

Stochastic Gradient Descent Convergence

In 2018, there have been several improvements:

- Under some models with high capacity, it can be proven that SGD will reach global optimum by showing it will reach zero training error.
- Neural networks can be easily modified so that the augmented version has no local minimums. Therefore, if such a network converges, it converged to a global minimum. However, the training process can still fail to converge by increasing the size of the parameters $\|\boldsymbol{\theta}\|$ beyond any limit.

Loss Function Visualization

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

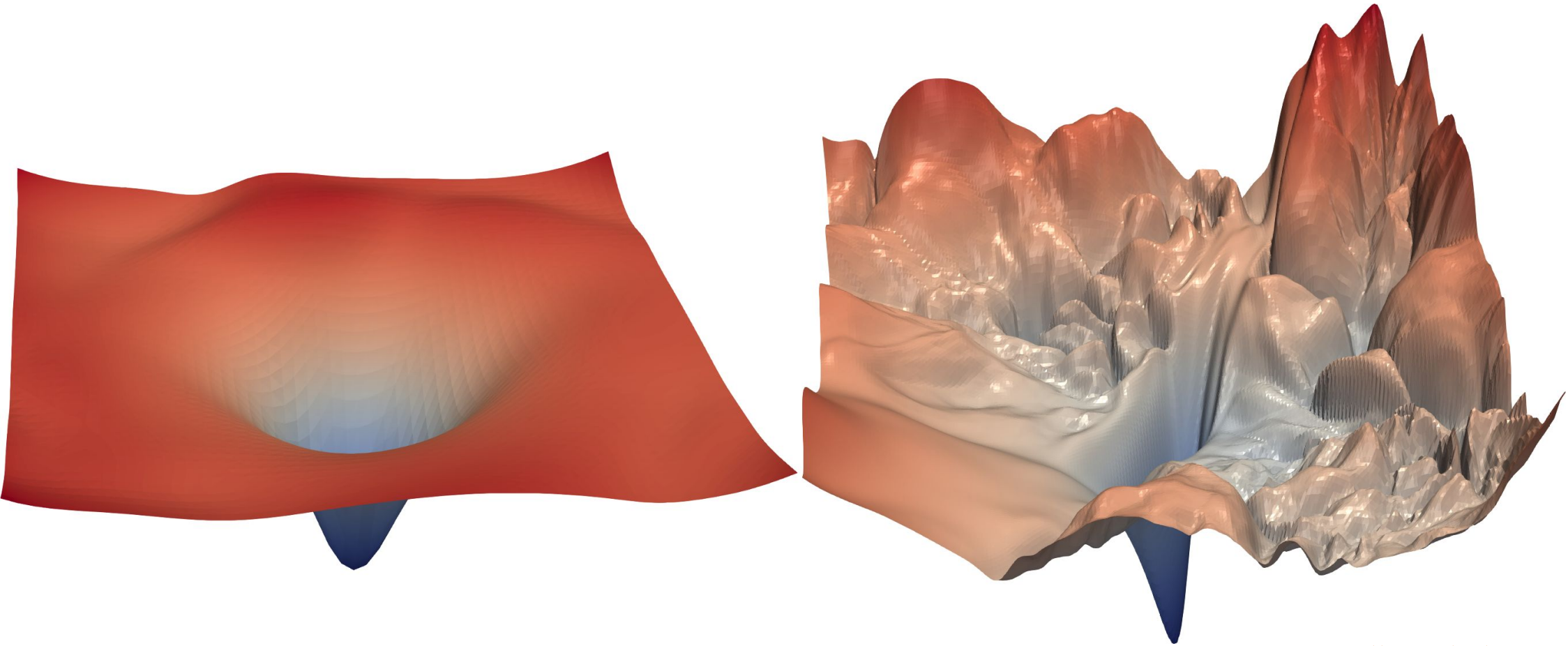
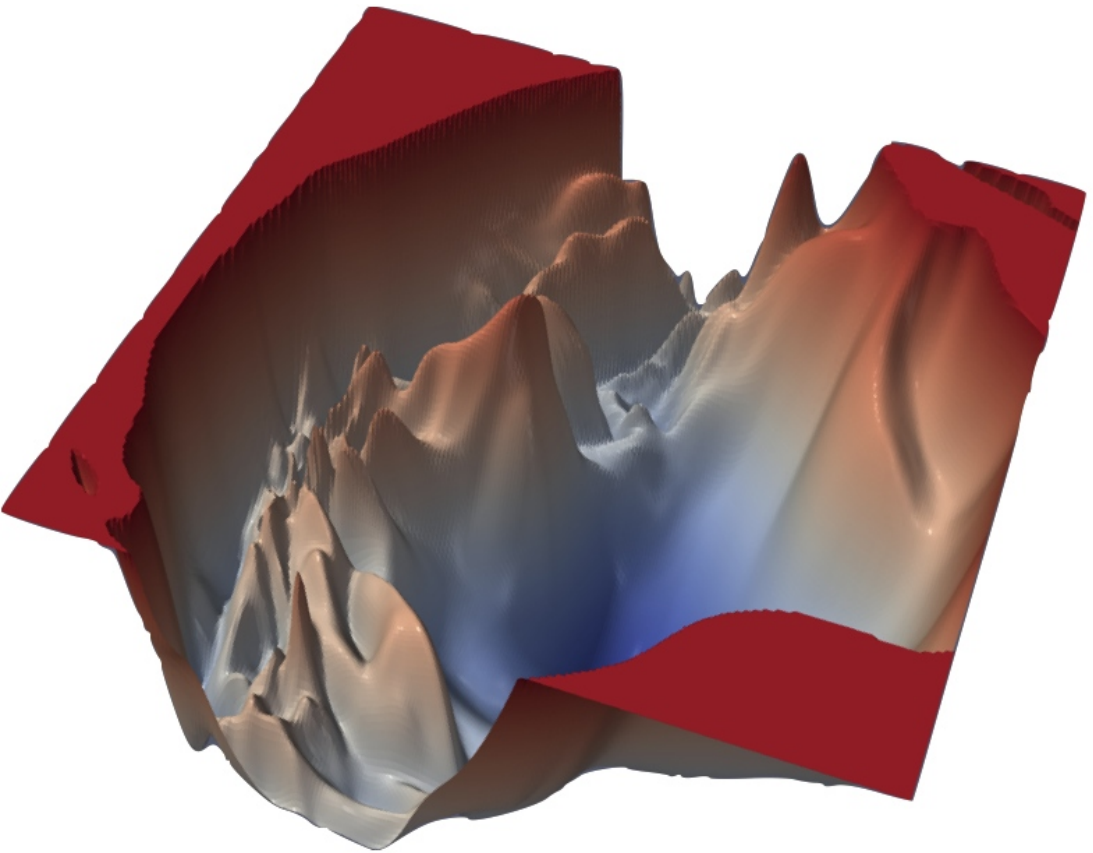


Figure 1 of "Visualizing the Loss Landscape of Neural Nets", https://arxiv.org/abs/1712.09913

Loss Function Visualization

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



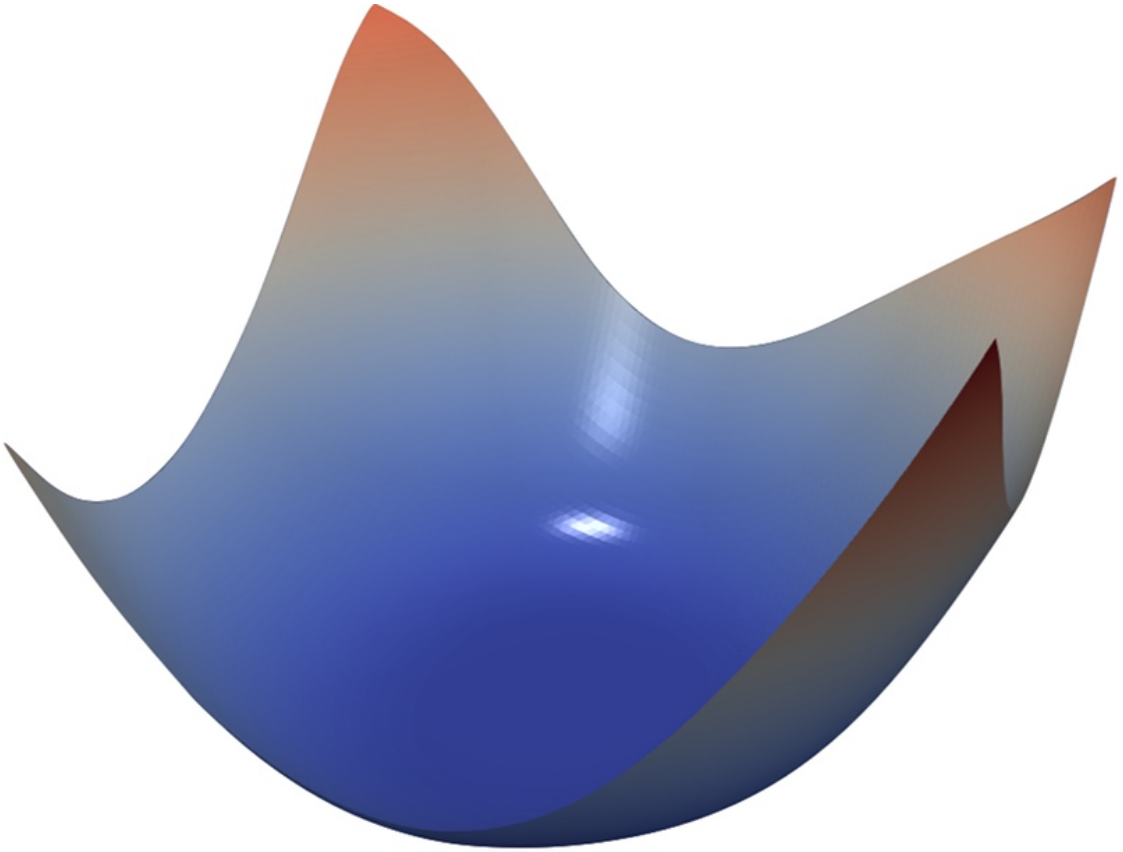
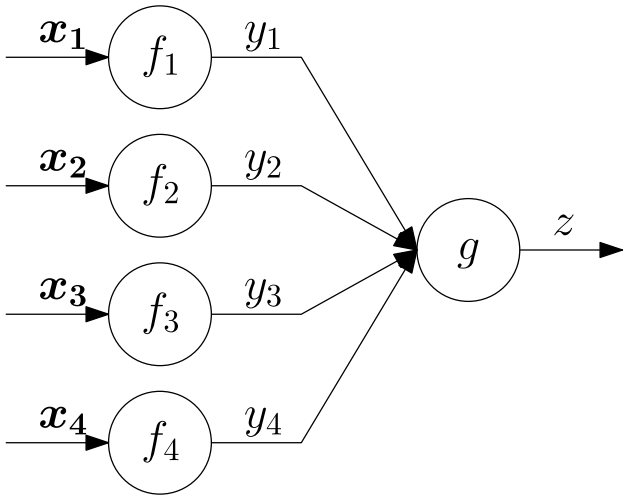


Figure 4 of "Visualizing the Loss Landscape of Neural Nets", https://arxiv.org/abs/1712.09913

Backpropagation

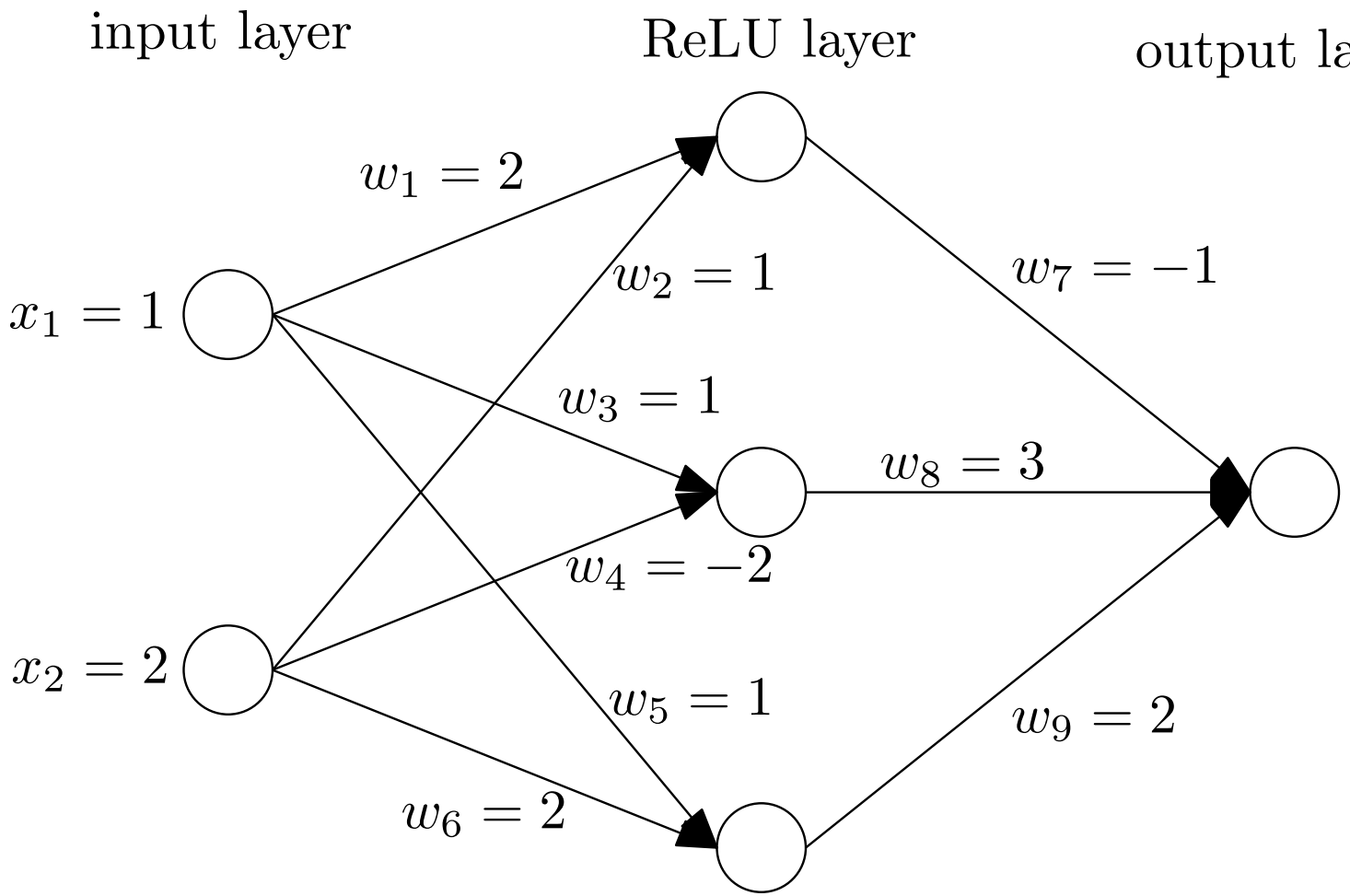
Assume we want to compute partial derivatives of a given loss function J and let $\frac{\partial J}{\partial z}$ be known.



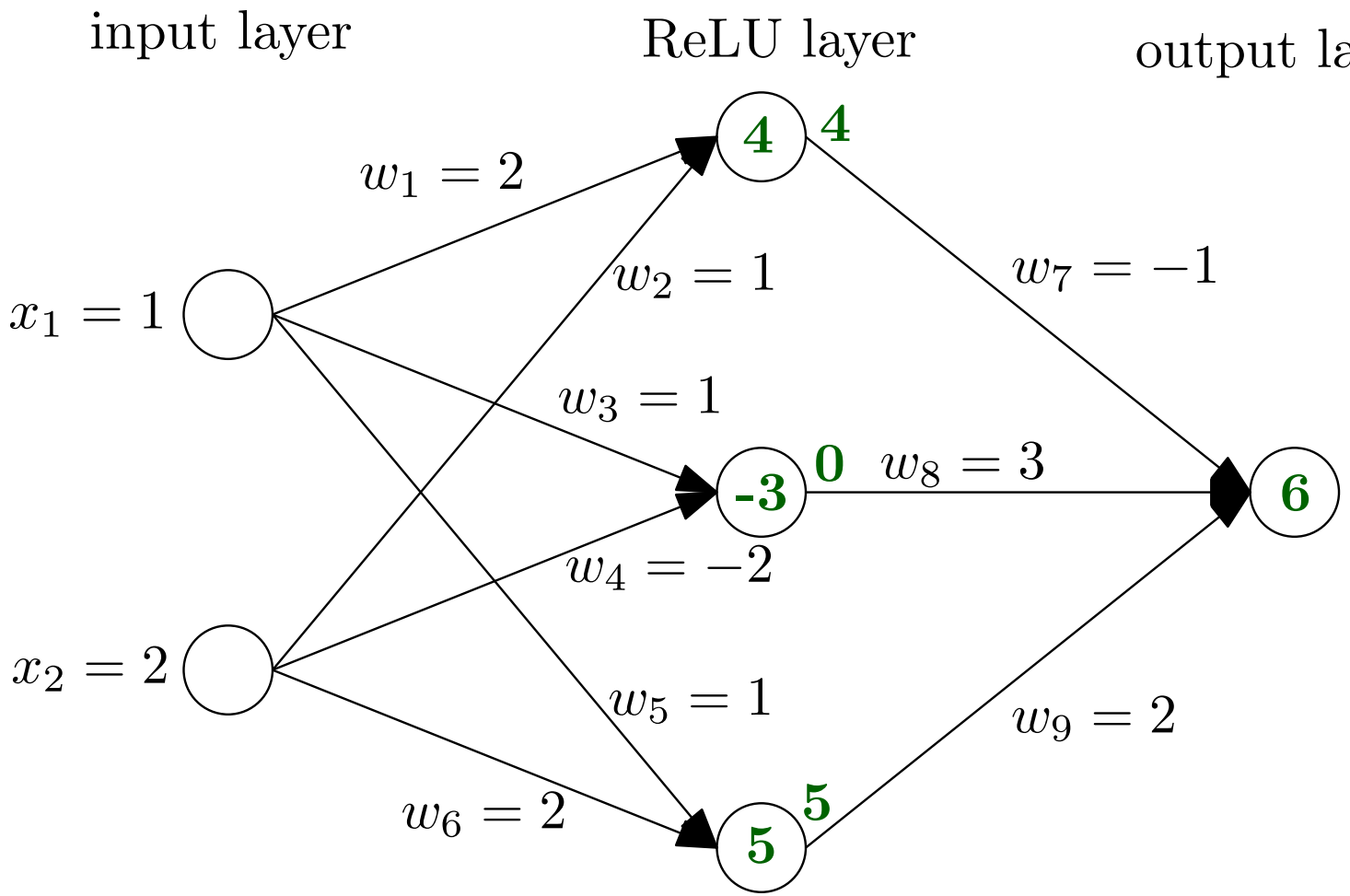
$$\frac{\partial J}{\partial y_i} = \frac{\partial J}{\partial z}\frac{\partial z}{\partial y_i} = \frac{\partial J}{\partial z}\frac{\partial g(\boldsymbol{y})}{\partial y_i}$$

$$\frac{\partial J}{\partial \boldsymbol{x}_i} = \frac{\partial J}{\partial z}\frac{\partial z}{\partial y_i}\frac{\partial y_i}{\partial \boldsymbol{x}_i} = \frac{\partial J}{\partial z}\frac{\partial g(\boldsymbol{y})}{\partial y_i}\frac{\partial f(\boldsymbol{x}_i)}{\partial \boldsymbol{x}_i}$$

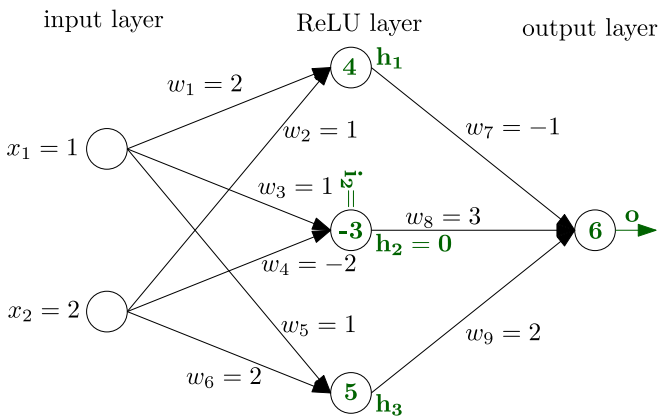
Backpropagation Example



Backpropagation Example



Backpropagation Example



$$\frac{\partial L}{\partial o} = 2(output - gold) = 6$$

$$\frac{\partial L}{\partial w_7} = \frac{\partial L}{\partial o}\frac{\partial o}{\partial w_7} = \frac{\partial L}{\partial o}h_1 = 24 \qquad \frac{\partial L}{\partial h_1} = \frac{\partial L}{\partial o}\frac{\partial o}{\partial h_1} = \frac{\partial L}{\partial o}w_7 = -6$$

$$\frac{\partial L}{\partial w_8} = \frac{\partial L}{\partial o}\frac{\partial o}{\partial w_8} = \frac{\partial L}{\partial o}h_2 = 0 \qquad \frac{\partial L}{\partial h_2} = \frac{\partial L}{\partial o}\frac{\partial o}{\partial h_2} = \frac{\partial L}{\partial o}w_8 = 18$$

$$\frac{\partial L}{\partial w_9} = \frac{\partial L}{\partial o}\frac{\partial o}{\partial w_9} = \frac{\partial L}{\partial o}h_3 = 30 \qquad \frac{\partial L}{\partial h_3} = \frac{\partial L}{\partial o}\frac{\partial o}{\partial h_3} = \frac{\partial L}{\partial o}w_9 = 12$$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial i_1}\frac{\partial i_1}{\partial w_1} = \frac{\partial L}{\partial i_1}x_1 = -6 \qquad \frac{\partial L}{\partial i_1} = \frac{\partial L}{\partial h_1}\frac{\partial h_1}{\partial i_1} = \frac{\partial L}{\partial h_1}1 = -6$$

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial i_1}\frac{\partial i_1}{\partial w_2} = \frac{\partial L}{\partial i_1}x_2 = -12 \qquad \frac{\partial L}{\partial i_2} = \frac{\partial L}{\partial h_2}\frac{\partial h_2}{\partial i_2} = \frac{\partial L}{\partial h_2}0 = 0$$

$$\frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial i_2}\frac{\partial i_2}{\partial w_3} = \frac{\partial L}{\partial i_2}x_1 = 0 \qquad \frac{\partial L}{\partial i_3} = \frac{\partial L}{\partial h_3}\frac{\partial h_3}{\partial i_3} = \frac{\partial L}{\partial h_3}1 = 12$$

$$\frac{\partial L}{\partial w_4} = \frac{\partial L}{\partial i_2}\frac{\partial i_2}{\partial w_4} = \frac{\partial L}{\partial i_2}x_2 = 0 \qquad \frac{\partial L}{\partial x_1} = \sum_j \frac{\partial L}{\partial i_j}\frac{\partial i_j}{\partial x_1} = 0$$

$$\frac{\partial L}{\partial w_5} = \frac{\partial L}{\partial i_3}\frac{\partial i_3}{\partial w_5} = \frac{\partial L}{\partial i_3}x_1 = 12 \qquad \frac{\partial L}{\partial x_2} = \sum_j \frac{\partial L}{\partial i_j}\frac{\partial i_j}{\partial x_2} = 18$$

$$\frac{\partial L}{\partial w_6} = \frac{\partial L}{\partial i_3}\frac{\partial i_3}{\partial w_6} = \frac{\partial L}{\partial i_3}x_2 = 24$$

This is meant to be frightening – you do **not** do this manually when training.

Forward Propagation

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

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

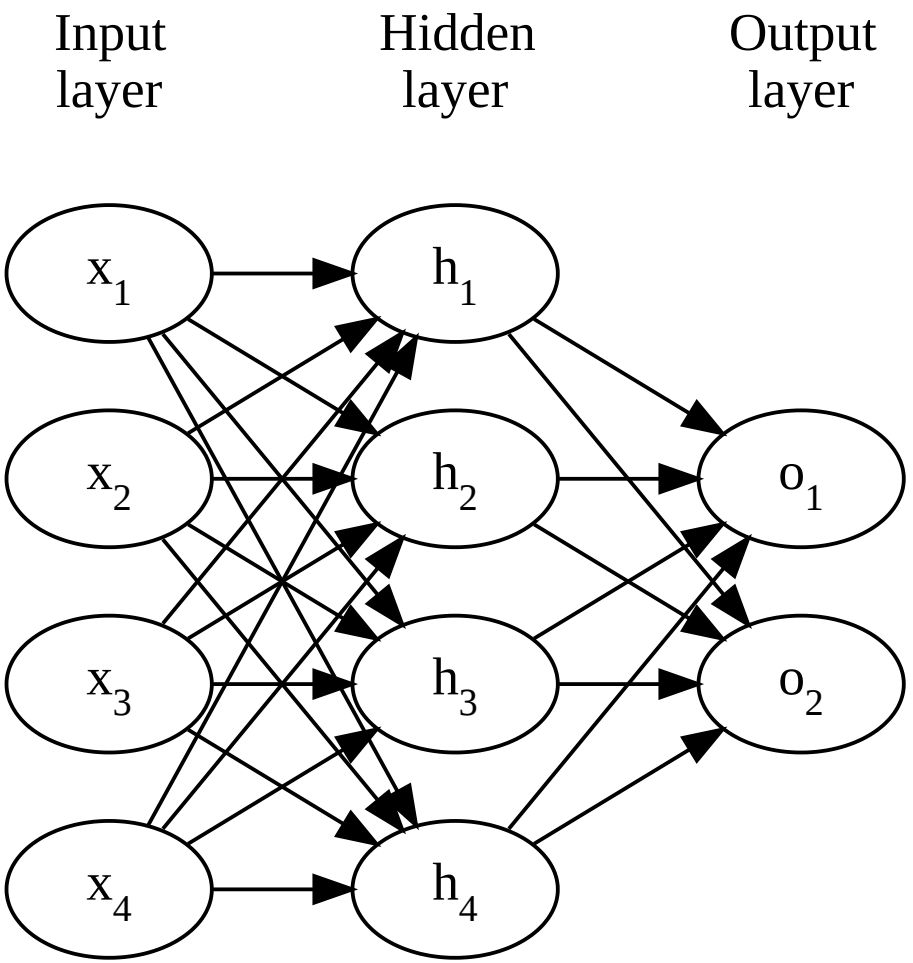
Simple Variant of Backpropagation

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 $\boldsymbol{g}$

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 $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^m$. Then $\frac{\partial \boldsymbol{f}(\boldsymbol{x})}{\partial \boldsymbol{x}}$ is a Jacobian. However, the backpropagation algorithm is analogous.

Neural Network Architecture à la '80s



Neural Network Architecture à la '80s

There is a weight on each edge, and an activation function f is performed on the hidden layers, and optionally also on the output layer.

$$h_i = f\left(\sum_j w_{i,j} x_j + b_i\right)$$

If the network is composed of layers, we can use matrix notation and write:

$$\boldsymbol{h} = f\left(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}\right)$$

Hidden Layers Derivatives

- σ:

$$\frac{d\sigma(x)}{dx} = \sigma(x) \cdot (1 - \sigma(x))$$

- $\tanh$:

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

- ReLU:

$$\frac{d\,\mathrm{ReLU}(x)}{dx} = \begin{cases} 1 & \text{if } x > 0 \\ \mathrm{NaN} & \text{if } x = 0;\ 0 \text{ is usually used} \\ 0 & \text{if } x < 0 \end{cases}$$

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 $(\boldsymbol{x}^{(i)}, y^{(i)})$
  - $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), y^{(i)})$
  - $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \boldsymbol{g}$

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 $\boldsymbol{\theta}$.

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

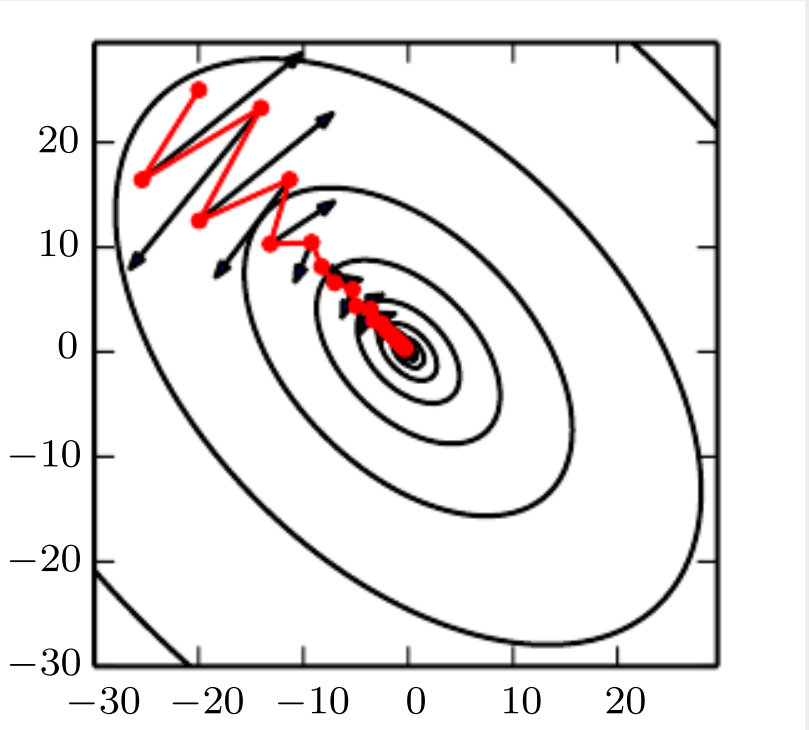


Figure 8.5 of "Deep Learning" book, https://www.deeplearningbook.org

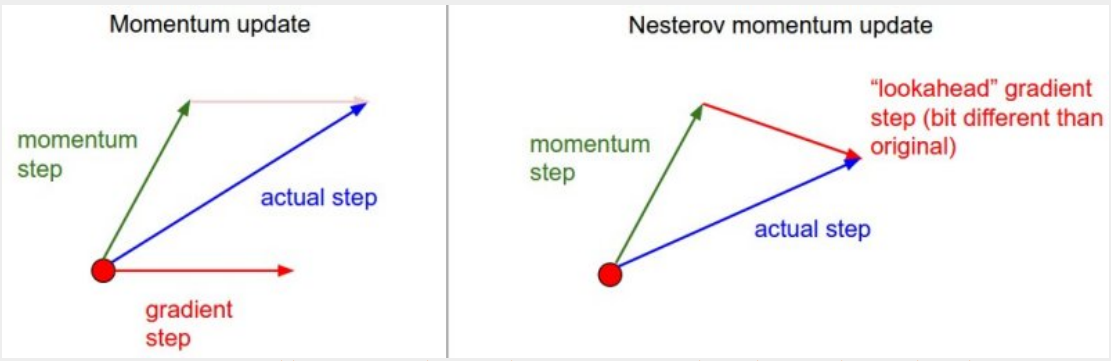
A nice writeup about momentum can be found on https://distill.pub/2017/momentum/.

SGD With Nesterov Momentum

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 $\boldsymbol{\theta}$.



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

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

AdaGrad (2011)

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

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

- The $\boldsymbol{g}^2$ and $\frac{\alpha}{\sqrt{\boldsymbol{r}} + \varepsilon} \boldsymbol{g}$ are computed element-wise.
- The $\boldsymbol{g}^2$ is sometimes also written as $\boldsymbol{g} \odot \boldsymbol{g}$.

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

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

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 $\boldsymbol{\theta}$.

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

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

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

Adam (2014)

Input: NN computing function $f(\boldsymbol{x}; \boldsymbol{\theta})$ with initial value of parameters $\boldsymbol{\theta}$.
Input: Learning rate α (default 0.001), constant ε (usually 10^{-7}).
Input: Momentum β_1 (default 0.9), momentum β_2 (default 0.999).
Output: Updated parameters $\boldsymbol{\theta}$.

- $\boldsymbol{s} \leftarrow 0$, $\boldsymbol{r} \leftarrow 0$, $t \leftarrow 0$
- Repeat until stopping criterion is met:
  - Sample a minibatch of m training examples $(\boldsymbol{x}^{(i)}, y^{(i)})$
  - $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), y^{(i)})$
  - $t \leftarrow t + 1$
  - $\boldsymbol{s} \leftarrow \beta_1 \boldsymbol{s} + (1 - \beta_1)\boldsymbol{g}$ *(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

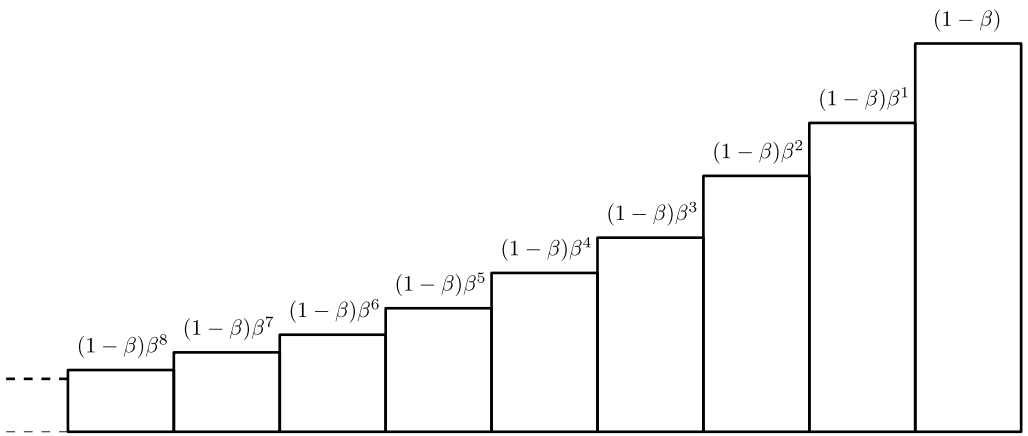
To allow analysis, we add indices to the update

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

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

After t steps, we have

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

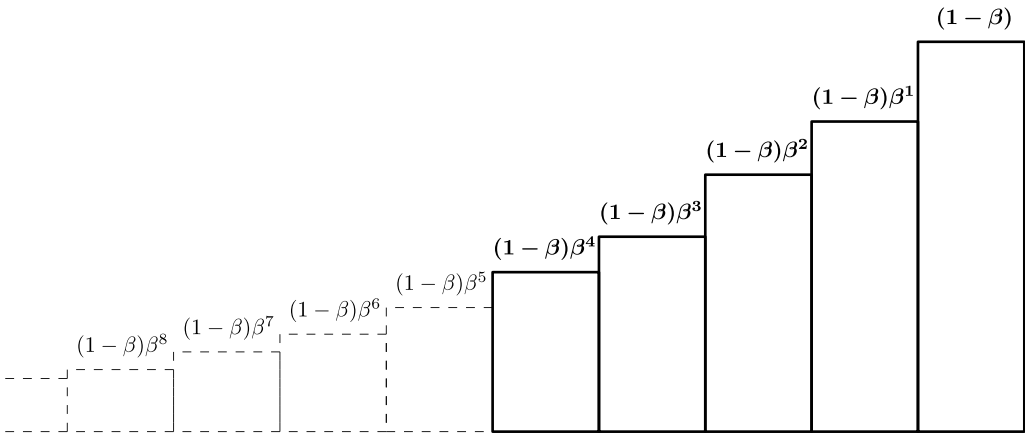


Because $\sum_{i=0}^{\infty} \beta_1^i = \frac{1}{1-\beta_1}$, $\boldsymbol{s}_\infty$ is computed as a weighted average of infinitely many elements.

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.

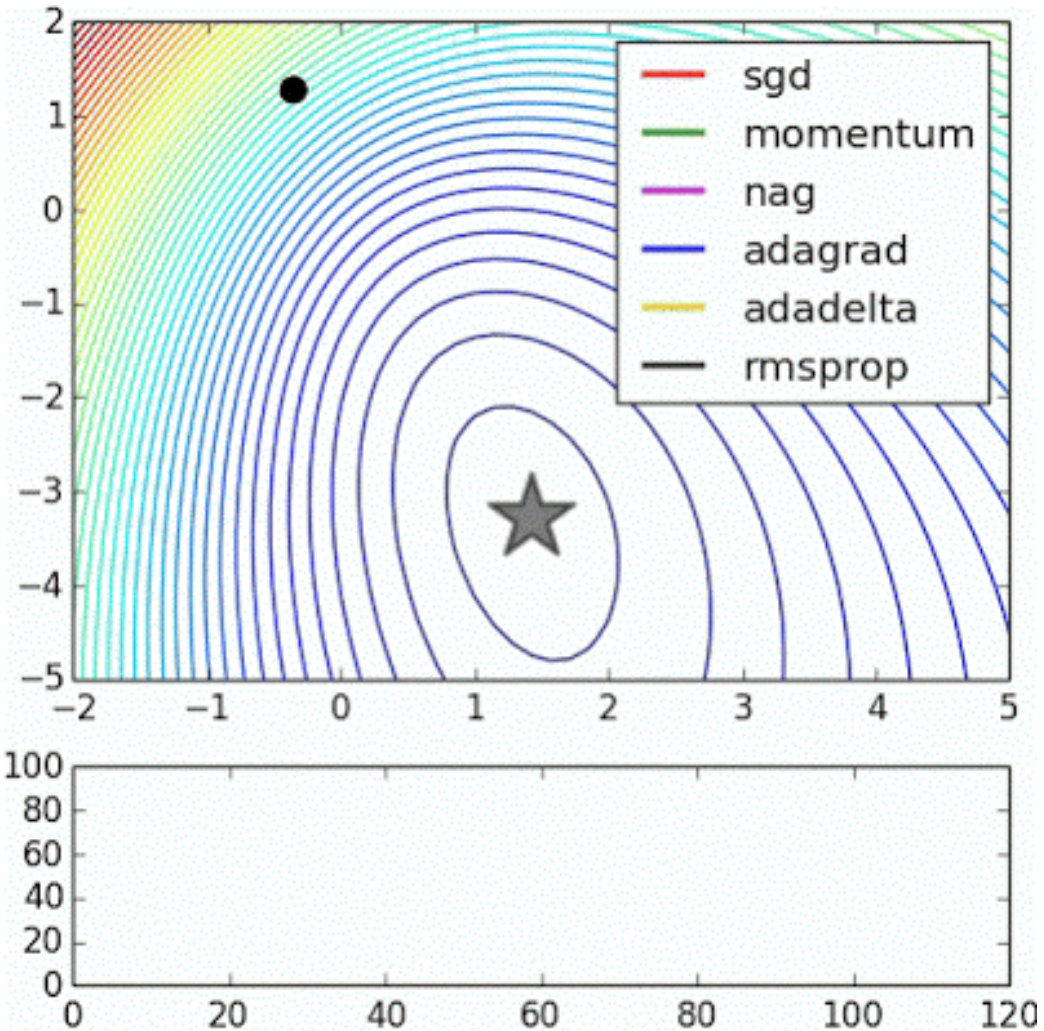


The sum of weights after t steps is

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

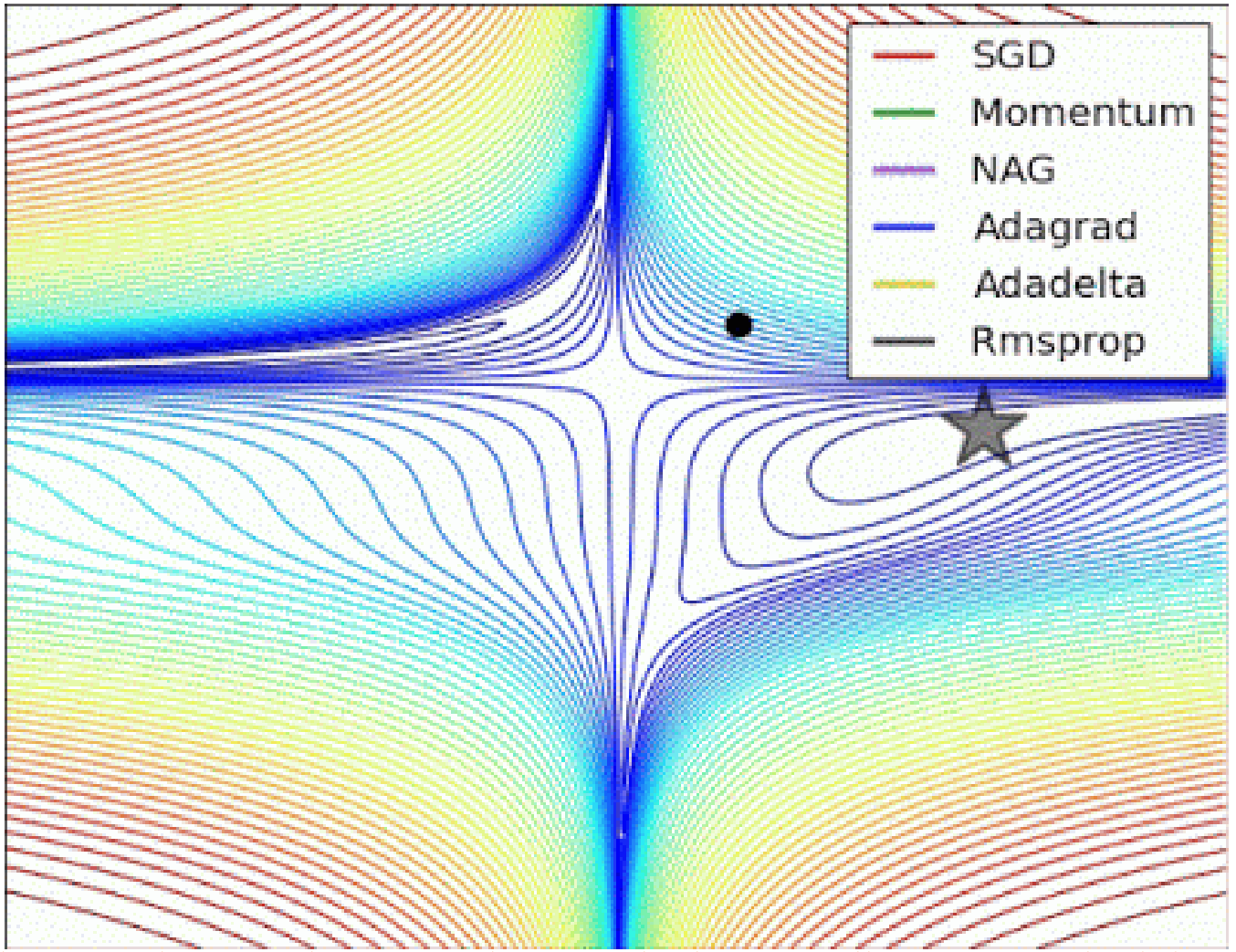
so we obtain an unbiased estimate by dividing $\boldsymbol{s}_t$ with $(1-\beta_1^t)$, and analogously for the correction of $\boldsymbol{r}$.

Adaptive Optimizers Animations



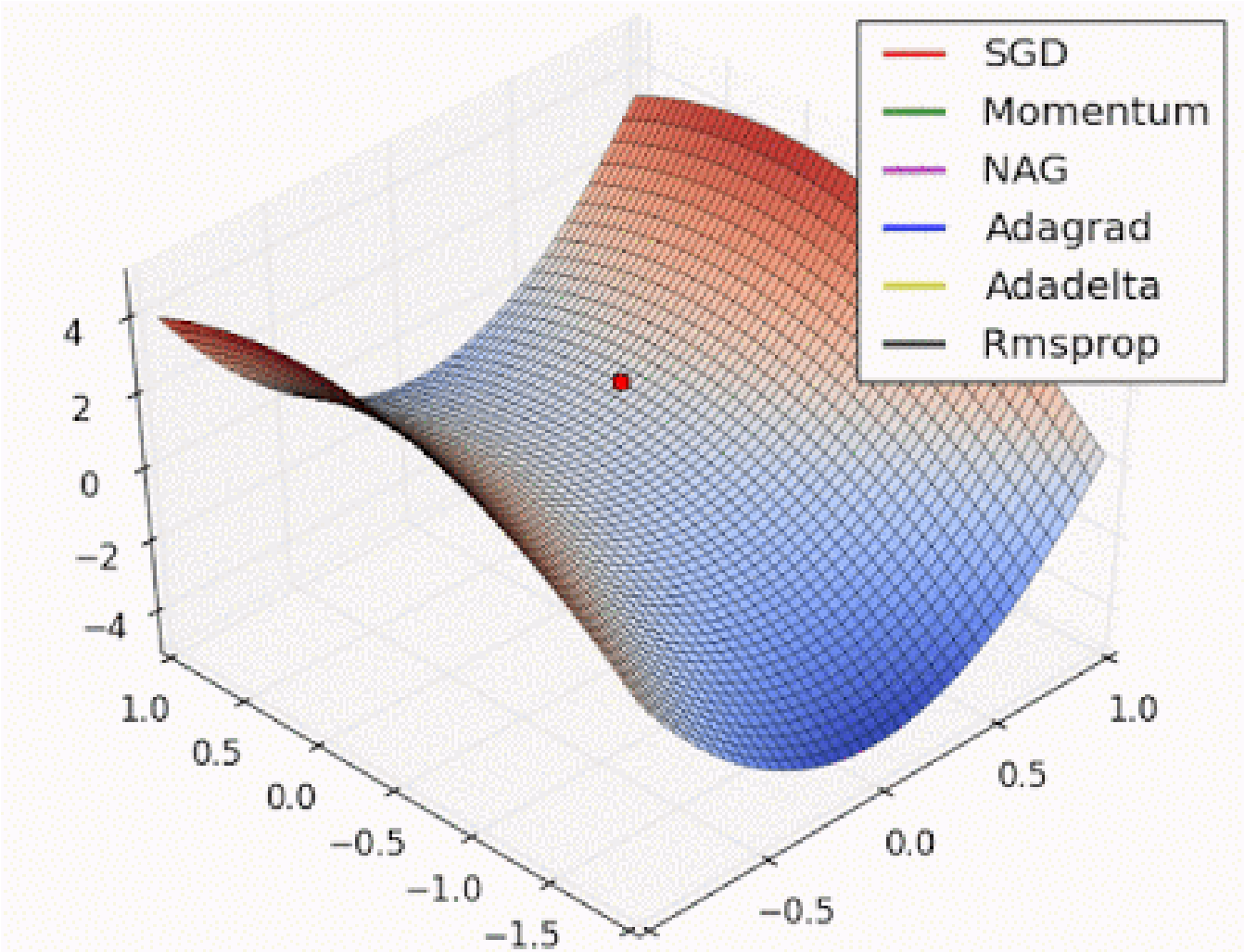
http://2.bp.blogspot.com/-q6l20Vs4P_w/VPmIC7sEhnI/AAAAAAAACC4/g3UOUX2r_yA/s400/ found at http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html

Adaptive Optimizers Animations



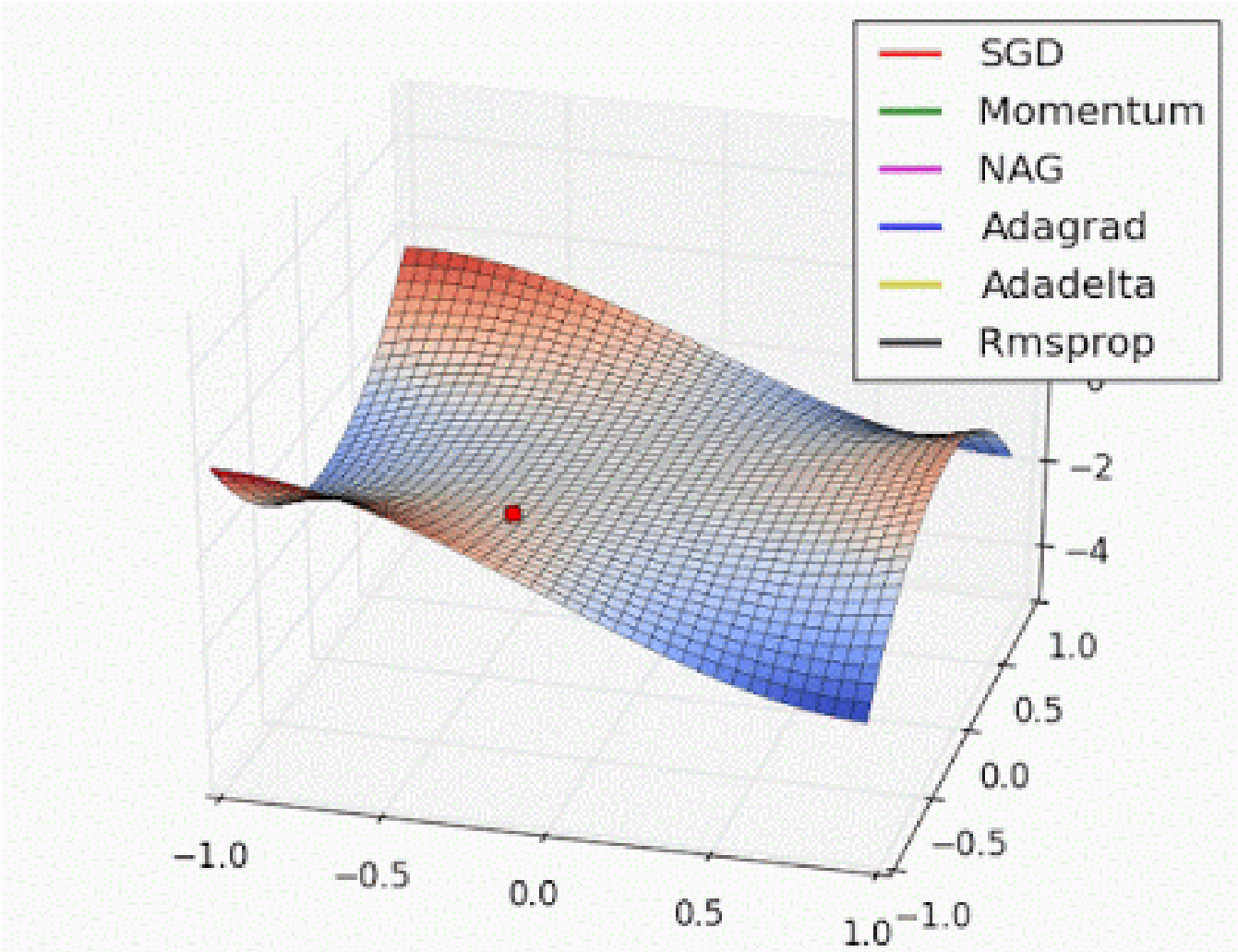
http://2.bp.blogspot.com/-L98w-SBmF58/VPmICIjKEKI/AAAAAAAACCs/rrFz3VetYmM/s400/ found at http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html

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

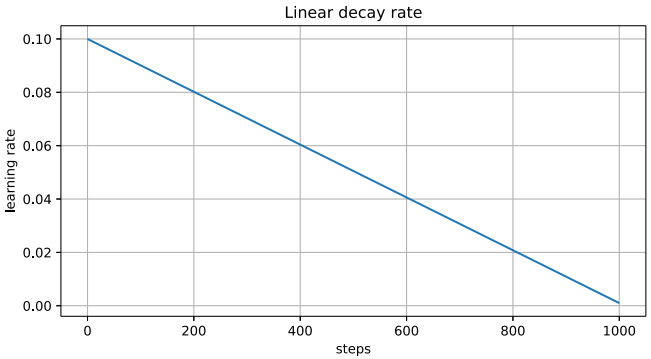


http://1.bp.blogspot.com/-K_X-yud8nj8/VPmIBxwGlsI/AAAAAAAACC0/JS-h1fa09EQ/s400/ found at http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html

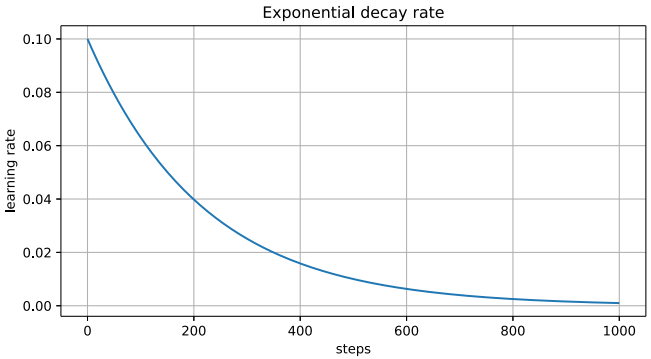
Learning Rate Schedules

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 t.
  - **Linear decay** uses $\alpha = \alpha_{\text{initial}} \cdot \left(1 - \frac{t}{\text{max steps}}\right)$ and has theoretical guarantees of convergence, but is usually too fast for deep neural networks.
  - **Inverse square root decay** uses $\alpha = \alpha_{\text{initial}} \cdot \frac{1}{\sqrt{t}}$ and is currently used by best machine translation models.



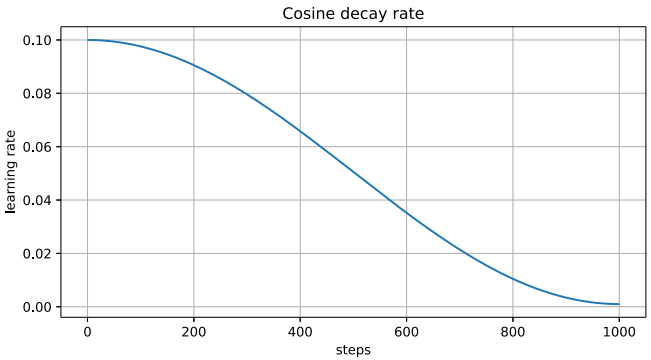
- **Exponential decay**: learning rate is multiplied by a constant each minibatch/epoch/several epochs.
  - $\alpha = \alpha_{\text{initial}} \cdot c^t$
  - Often used for convolutional networks (image recognition etc.).



Learning Rate Schedules

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

$$\frac{1}{2}\left(1+\cos\left(\pi\cdot\frac{t}{\text{max steps}}\right)\right)$$



- Cyclic restarts, warmup, ...

The `tf.optimizers.schedules` offers several such learning rate schedules, which can be passed to any Keras optimizer directly as a learning rate.

- `tf.optimizers.schedules.PolynomialDecay`
- `tf.optimizers.schedules.ExponentialDecay`
- `tf.optimizers.schedules.CosineDecay`