Optimization for Deep Learning

Vaibhav Rajan
Department of Information Systems & Analytics
School of Computing, National University of Singapore

Optimization

$\mathbf{x}$: vector of decision variables
$f, g_1, g_2, \ldots, g_n$: functions
$b_1, \ldots, b_n$: constants

- Minimize $f(\mathbf{x})$
- s.t. $g_i(\mathbf{x}) \geq b_i$ for $i = 1, \ldots, n$

Maximization is equivalent: maximizing $f(\mathbf{x})$ done by minimizing $-f(\mathbf{x})$

If there are no constraints: Unconstrained Optimization

f: Objective function, ML: Cost, Loss or Error function

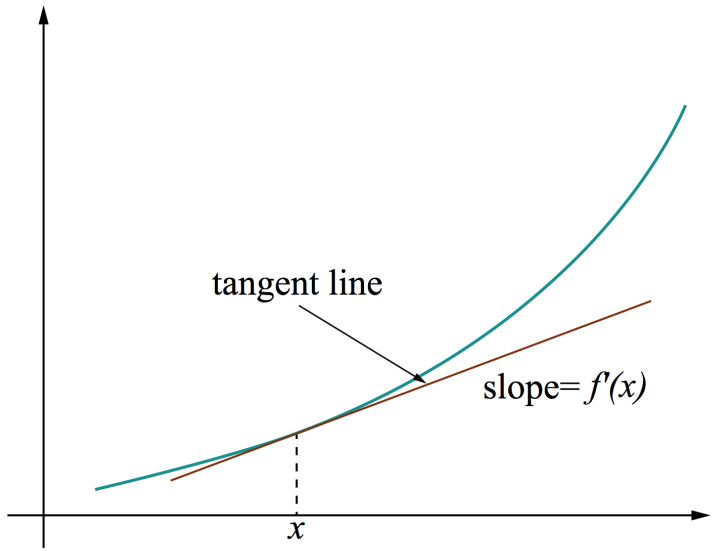
$\mathbf{x}^* = \arg\min f(\mathbf{x})$

Calculus 101

- Gradient
- Curvature
- Critical Point

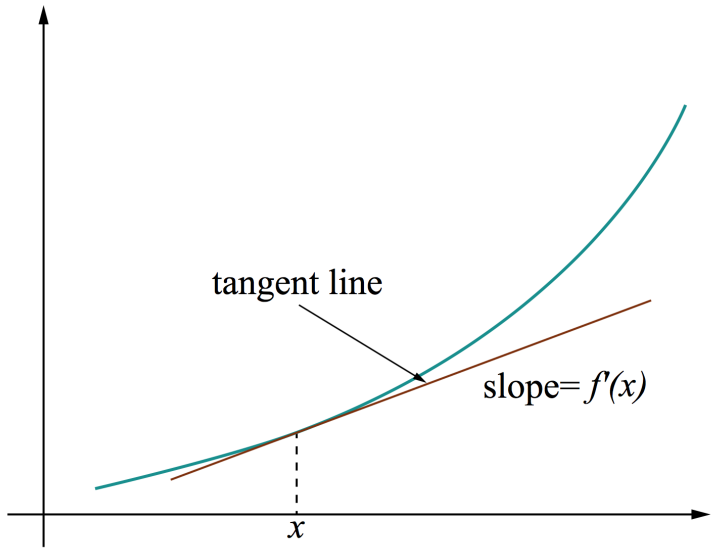
Derivatives

$f(x)$: function of a single variable x.



The derivative of a function of a single variable at a chosen input value (when it exists) is the slope of the tangent line to the graph of the function at that point.

Derivatives



$$f(x+\epsilon) \approx f(x) + \epsilon f'(x)$$

So, the derivative tells us which points in the neighborhood of x will help in increasing/decreasing the value of $f(x)$: to increase move in the direction of the derivative.

Differentiation Rules

f, g: real-valued functions of x; a, b: real numbers; f': derivative of f wrt x

Name	Function	Derivative
Linearity	$h = af + bg$	$h' = af' + bg'$
Sum	$h = f + g$	$h' = f' + g'$
Product	$h = f(x)g(x)$	h' $= f'(x)g(x) + g'(x)f(x)$
Chain	$h = f(g(x))$	$f'(g(x))g'(x)$
Polynomial	$f = x^a$	$f' = ax^{a-1}$
Exponential	$f = e^{ax}$	$f' = ae^{ax}$
Logarithm	$f = \log x$	$f' = \frac{1}{x}$

Partial Derivatives

f: function of n variables

$$f(\mathbf{x}) \equiv f(x_1, x_2, \ldots, x_n)$$

The partial derivative of f w.r.t. x_i is defined by the following limit (when it exists):

$$\frac{\partial f}{\partial x_i} = \lim_{h \to 0} \frac{f(x_1, \ldots, x_i, x_i + h, x_{i+1}, \ldots, x_n)}{h}$$

Gradient

The gradient vector of f is given by:

$$\nabla f = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix}$$

The gradient points along the direction in which the function increases most rapidly.

Second Derivative

The second derivative of an n-variable function is given by:

$$\frac{\partial}{\partial x_i}\left(\frac{\partial f}{\partial x_j}\right) \quad i = 1, \ldots, n;\ j = 1, \ldots, n$$

It is usually written as:

$$\frac{\partial^2 f}{\partial x_i \partial x_j},\ i \neq j; \quad \frac{\partial^2 f}{\partial x_i^2},\ i = j$$

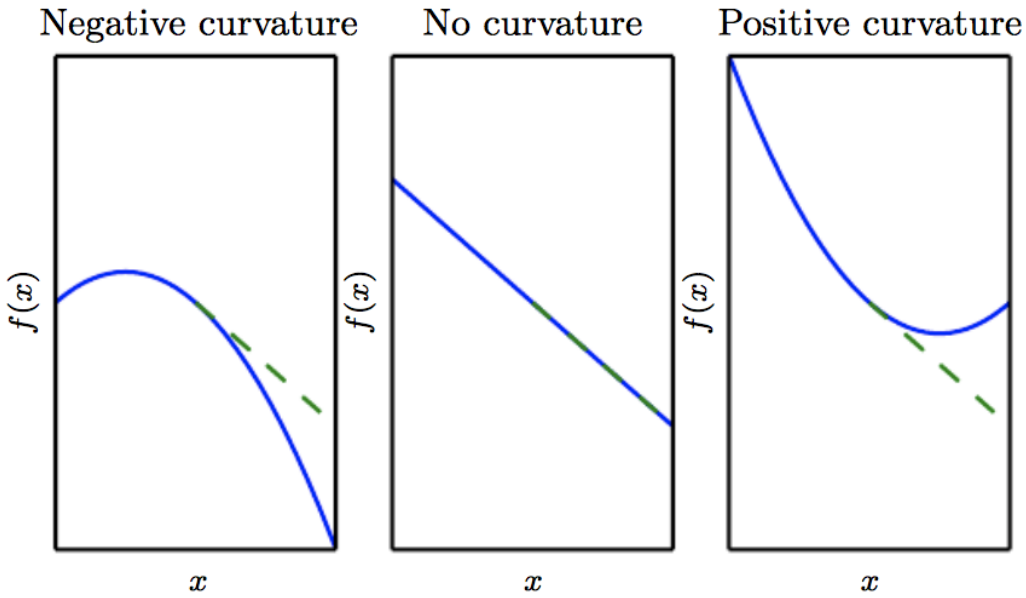
Second Derivative

The second derivative tells us how the first derivative changes as we vary the input.

We can think of the second derivative as measuring **curvature**.

This is important because it tells us whether a gradient step will cause as much of an improvement as we would expect based on the gradient alone.

Second Derivative



Dashed line: value of the function we expect based on the gradient alone.

With negative curvature, the cost function actually decreases faster than the gradient predicts. With no curvature, the gradient predicts the decrease correctly. With positive curvature, the function decreases more slowly than expected and eventually begins to increase.

Hessian

All the n^2 second partial derivatives of a function are represented by a square, symmetric matrix called the Hessian matrix of $f(\mathbf{x})$

$$\mathbf{H}_f(\mathbf{x}) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & & \vdots \\ \frac{\partial^2 f}{\partial x_1 \partial x_n} & \cdots & \frac{\partial^2 f}{\partial x_1^2} \end{pmatrix}$$

Critical Point

When all first-order partial derivatives at a point are zero (i.e. $\nabla f = 0$) then the point is said to be a stationary or critical point.

A critical point can correspond to a minimum, maximum or saddle point of the function.

At critical points, derivatives provide no information on the direction to move.

Critical Point

Local Minimum (Maximum): $f(\mathbf{x})$ is lower (higher) than at all neighboring points. Saddle points: neither local maxima or minima.

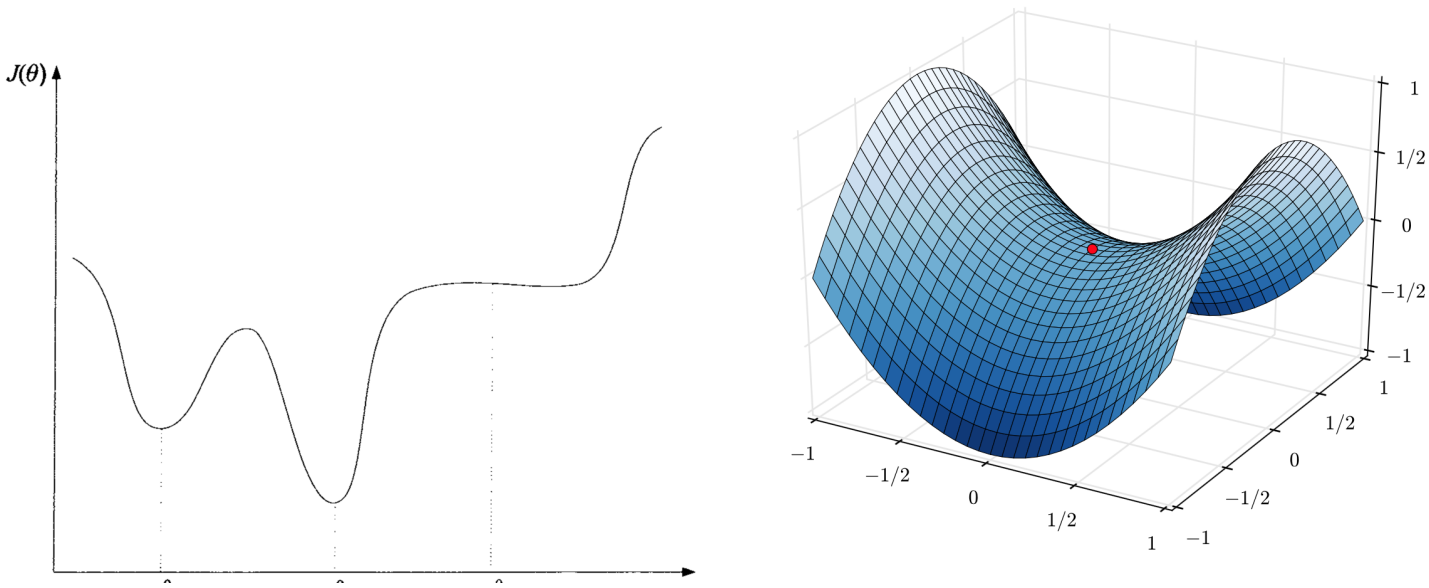


Figure from: https://en.wikipedia.org/wiki/Saddle_point

Critical Point

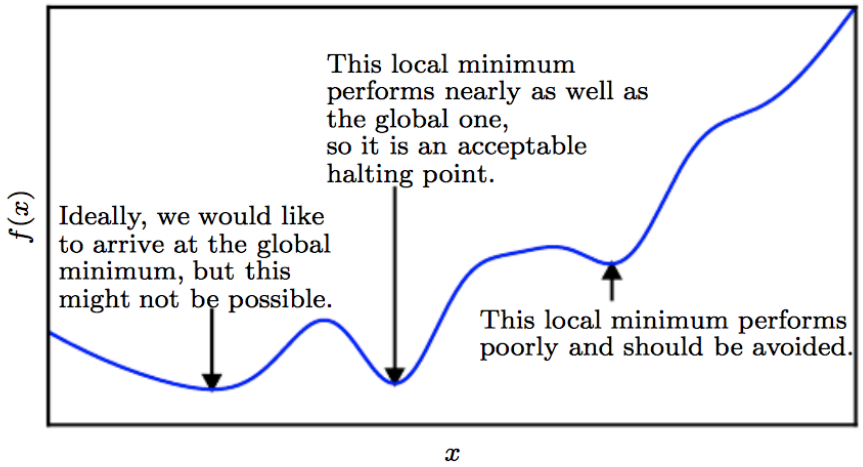
The second derivative can be used to determine whether a critical point is a local maximum, a local minimum, or a saddle point.

- Local Minimum: $f'(x) = 0,\ f''(x) > 0$
- Local Maximum: $f'(x) = 0,\ f''(x) < 0$
- Saddle Point or Flat region: $f'(x) = 0,\ f''(x) = 0$

Using the eigendecomposition of the Hessian matrix, we can generalize the second derivative test to multiple dimensions.

Global Optima

A point $\mathbf{x}$ that obtains the absolute lowest value of $f(\mathbf{x})$ is a global minimum.



In deep learning, we optimize functions that may have many local minima and many saddle points surrounded by very flat regions. This makes optimization difficult, especially with multidimensional inputs. We settle for finding a value of f that is very low but not necessarily minimal in any formal sense.

Constrained Optimization

Minimize $f(\mathbf{x})$ subject to condition $c(\mathbf{x})$.

Key Idea: Use Lagrangian function to convert to an unconstrained optimization problem.

- Lagrangian function $\mathcal{L}(\mathbf{x}, \lambda) = f(\mathbf{x}) + \lambda c(\mathbf{x})$
- Critical point of $\mathcal{L}(\mathbf{x}, \lambda)$ wrt $\mathbf{x}$ and λ
- Gives both critical point $\mathbf{x}^*$ that minimizes $f(\mathbf{x})$ and value of λ
- Scalar λ may be eliminated through the equations if its value is not needed

LP, ILP, QP

- Linear Programming (LP): linear objective, linear constraints.
- Integer Linear Programming (ILP): linear program + some/all decision variables are constrained to be integers.
- Quadratic Programming (QP): quadratic objective, linear constraints.

Illustration: Least Squares Estimates for Simple Linear Regression

Simple Linear Regression

Given training data: X_i, Y_i for $i = 1, \ldots, n$, find the best-fit line

$$\mathbf{Y} = \hat{\beta}_0 + \hat{\beta}_1 \mathbf{X}$$

such that the sum of squared errors $\sum_i (Y_i - \hat{Y}_i)^2$ is minimized.

Simple Linear Regression

Given training data: X_i, Y_i for $i = 1, \ldots, n$, find the best-fit line

$$\mathbf{Y} = \hat{\beta}_0 + \hat{\beta}_1 \mathbf{X}$$

such that the sum of squared errors $\sum_i (Y_i - \hat{Y}_i)^2$ is minimized.

- Function to be minimized: $f(\beta_0, \beta_1) = \sum_i (Y_i - \hat{Y}_i)^2$
- Unknown parameters: β_0, β_1

Simple Linear Regression

Minimize: $f(\beta_0, \beta_1) = \sum_i (Y_i - \hat{Y}_i)^2 = \sum_i (Y_i - \beta_0 - \beta_1 X_i)^2$

f will be minimized at the values of β_0, β_1 for which $\frac{\partial f}{\partial \beta_0} = 0$ and $\frac{\partial f}{\partial \beta_1} = 0$.

Simple Linear Regression

Minimize: $f(\beta_0, \beta_1) = \sum_i (Y_i - \hat{Y}_i)^2 = \sum_i (Y_i - \beta_0 - \beta_1 X_i)^2$

f will be minimized at the values of β_0, β_1 for which $\frac{\partial f}{\partial \beta_0} = 0$ and $\frac{\partial f}{\partial \beta_1} = 0$.

$$\frac{\partial f}{\partial \beta_0} = \sum_i -2(Y_i - \beta_0 - \beta_1 X_i) = 2(n\beta_0 + \beta_1 \sum_i X_i - \sum_i Y_i) = 0$$

Divide by 2, use $\sum_i X_i = n\bar{X}$ and solve for β_0:

$$\beta_0 = \bar{Y} - \beta_1 \bar{X}$$

The intercept β_0 is set such that the line must pass through the means, $\bar{X}$, $\bar{Y}$ of X_i and Y_i respectively.

Simple Linear Regression

Minimize: $f(\beta_0, \beta_1) = \sum_i (Y_i - \hat{Y}_i)^2 = \sum_i (Y_i - \beta_0 - \beta_1 X_i)^2$

f will be minimized at the values of β_0, β_1 for which $\frac{\partial f}{\partial \beta_0} = 0$ and $\frac{\partial f}{\partial \beta_1} = 0$.

$$\frac{\partial f}{\partial \beta_1} = \sum_i -2X_i(Y_i - \beta_0 - \beta_1 X_i) = \sum_i -2(X_i Y_i - \beta_0 X_i - \beta_1 X_i^2) = 0$$

Substitute expression for β_0, divide by 2 and use $\sum_i X_i = n\bar{X}$:

$$\sum_i (X_i Y_i - X_i \bar{Y} + \beta_1 X_i \bar{X} - \beta_1 X_i^2) = \sum_i (X_i Y_i - X_i \bar{Y}) - \beta_1 \sum_i (X_i^2 - X_i \bar{X}) = 0$$

$$\beta_1 = \frac{\sum_i (X_i Y_i - X_i \bar{Y})}{\sum_i (X_i^2 - X_i \bar{X})} = \frac{\sum_i (X_i Y_i) - n\bar{X}\bar{Y}}{\sum_i (X_i^2) - n\bar{X}^2}$$

Simple Linear Regression

Minimize: $f(\beta_0, \beta_1) = \sum_i (Y_i - \hat{Y}_i)^2 = \sum_i (Y_i - \beta_0 - \beta_1 X_i)^2$

f will be minimized at the values of β_0, β_1 for which $\frac{\partial f}{\partial \beta_0} = 0$ and $\frac{\partial f}{\partial \beta_1} = 0$.

$$\beta_1 = \frac{\sum_i (X_i Y_i) - n\bar{X}\bar{Y}}{\sum_i (X_i^2) - n\bar{X}^2}$$

$$\beta_0 = \bar{Y} - \beta_1 \bar{X}$$

Multiple Linear Regression

Minimize:
$f(\beta_0, \ldots, \beta_p) = \sum_i (Y_i - \hat{Y}_i)^2 = \sum_i (Y_i - \beta_0 - \beta_1 X_{i1} - \ldots - \beta_p X_{ip})^2$

$\beta_0, \ldots, \beta_p$ can be found using matrix derivatives.

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{Y}$$

Proof can be found here.

Optimization in Machine Learning

Model Training

Unconstrained Optimization: Regression, Neural Networks,...

Constrained Optimization: PCA, SVM ...

Difference from Pure Optimization

Pure Optimization: minimizing $f(\mathbf{x})$ is the final goal

E.g. in problems related to budgeting, scheduling, financial planning, combinatorial optimization etc.

ML acts indirectly: during training we optimize performance metric on training data

But, we want generalization on unseen test data

Gradient Descent

Gradient Descent

Algorithm to find local minimum of a function $f(\mathbf{x})$.

- Initialize $\mathbf{x}^{(t)}$ for iteration $t = 0$
- Repeat until convergence: $\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \epsilon \nabla f(\mathbf{x}^{(t)})$.

ϵ: step size or learning rate

$-\nabla f(\mathbf{x}_t)$: direction of steepest descent

Superscript (t): denotes t^{th} iteration (not exponent)

Gradient Descent

- Initialize $\mathbf{x}^{(t)}$ for iteration $t = 0$
- Repeat until convergence: $\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \epsilon \nabla f(\mathbf{x}^{(t)})$.

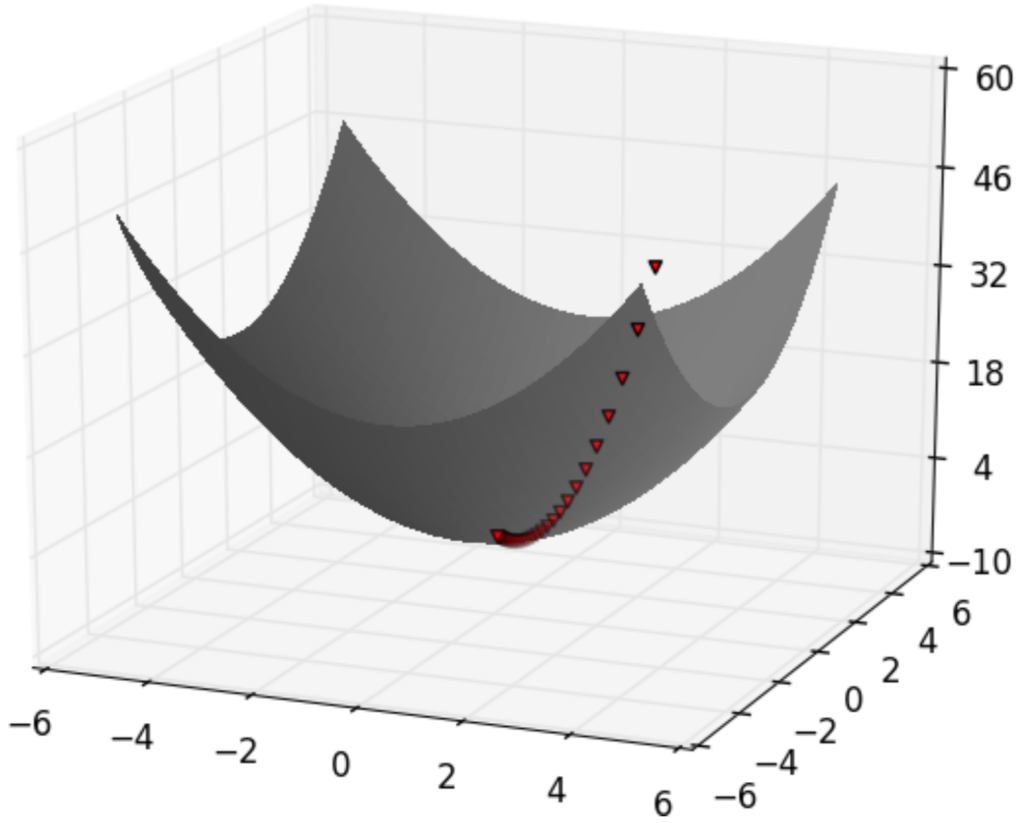
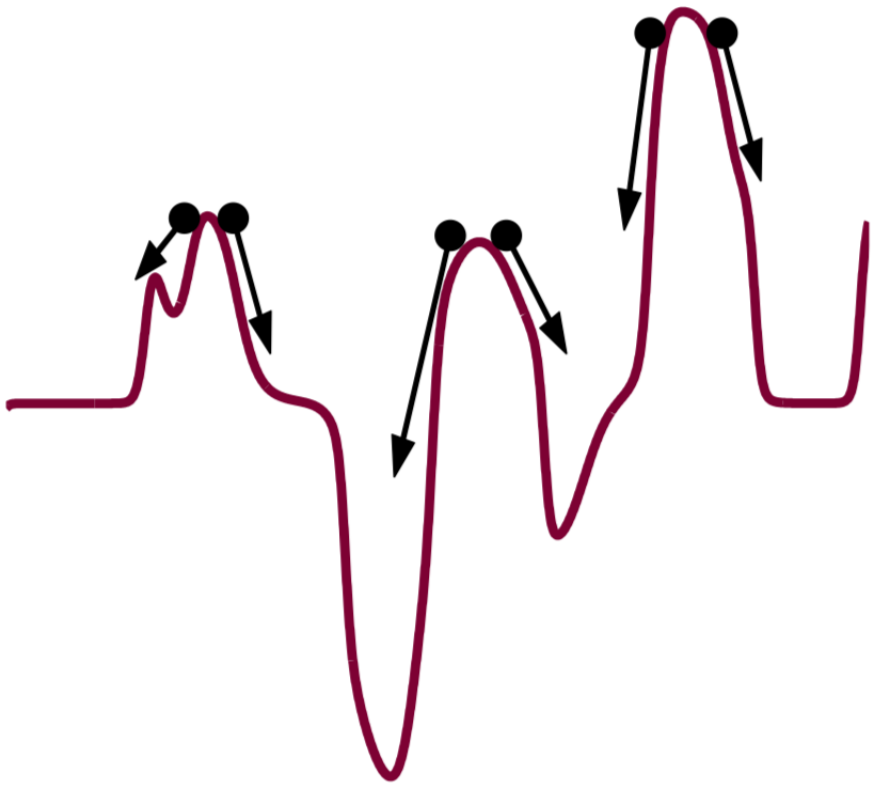
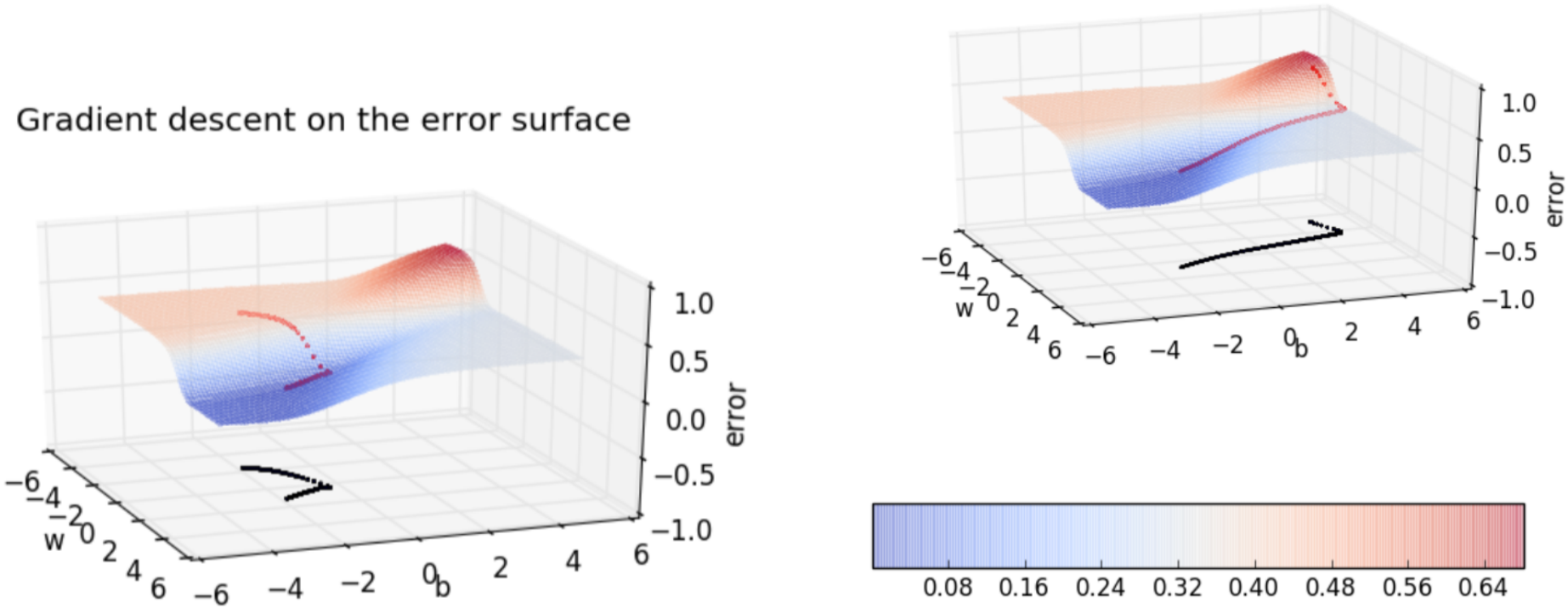


Figure from Joel Grus, Data Science from Scratch

Initialization Matters!



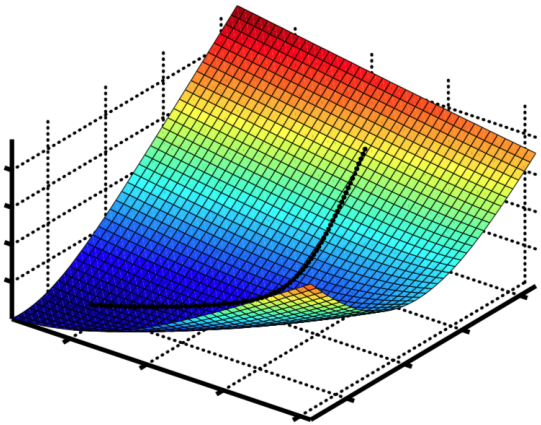
Initialization Matters!

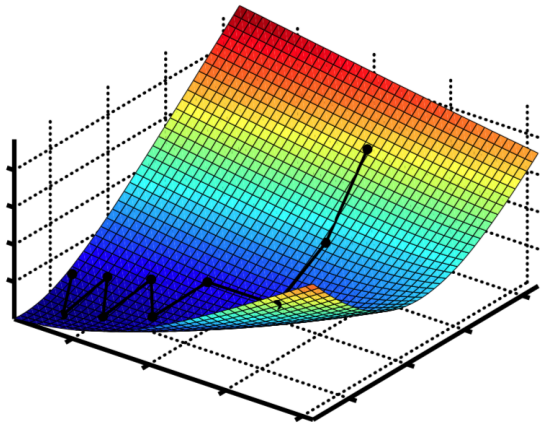


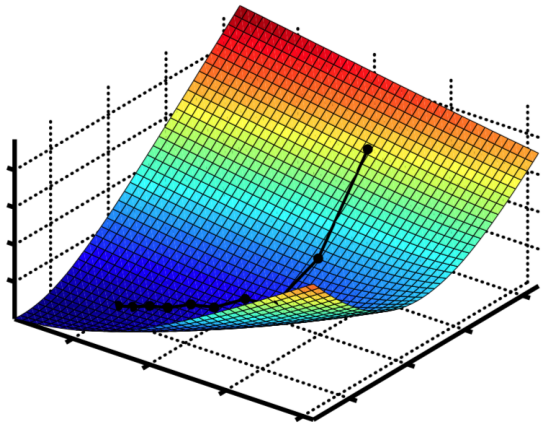
In areas where the slope is gentle the gradients (hence, updates) are small whereas in the areas where the slope is steep the updates are large.

Learning Rate

- Too small: convergence is too slow
- Too large: may overshoot the minimum







Gradient Descent for Logistic Regression (Training a Neuron)

Logistic Regression

Simple Logistic Regression:

- Odds: $\frac{p(Y)}{1-p(Y)} = e^{\beta_0+\beta_1 X}$
- Logit: $\log\left(\frac{p(Y)}{1-p(Y)}\right) = \beta_0 + \beta_1 X$

Multiple Logistic Regression:

- $\log\left(\frac{p(y)}{1-p(y)}\right) = \beta_0 + \beta_1 x_1 + \ldots + \beta_p x_p$
- $p(y=1|\boldsymbol{\beta}, \mathbf{x}) = \frac{e^{\beta_0+\beta_1 x_1+\ldots+\beta_p x_p}}{1+e^{\beta_0+\beta_1 x_1+\ldots+\beta_p x_p}} = \frac{e^{\boldsymbol{\beta}\mathbf{x}}}{1+e^{\boldsymbol{\beta}\mathbf{x}}} = \frac{1}{1+e^{-\boldsymbol{\beta}\mathbf{x}}}$

where we define $\mathbf{x} = (1\ x_1\ \ldots x_p), \boldsymbol{\beta} = (\beta_0\ \beta_1 \ldots \beta_p)^T$

Logistic Regression

$$h_{\boldsymbol{\beta}}(\mathbf{x}) = p(y = 1|\boldsymbol{\beta}, \mathbf{x}) = \frac{1}{1 + e^{-\boldsymbol{\beta}\mathbf{x}}}$$

Probability of label $y = 1$ for a feature vector $\mathbf{x}$.

Logistic or sigmoid function: $\sigma(z) = \frac{1}{1+e^{-z}}$

Likelihood

General framework for machine learning.

Idea: Given dataset D and model M (with its distributional assumptions), Likelihood: $p(D|M)$.

Used to estimate model parameters through optimization: Find those parameters that maximize the likelihood.

Logistic Regression: Data consists of feature vectors $\mathbf{X}$, labels $\mathbf{y}$, we consider the likelihood $\mathcal{L}(\beta) = p(\mathbf{y}|\mathbf{X}, \beta)$.

Likelihood

When $y = 1, p(y = 1|\boldsymbol{\beta}, \mathbf{x}) = h_{\boldsymbol{\beta}}(\mathbf{x})$

$$h_{\boldsymbol{\beta}}(\mathbf{x}) = p(y = 1|\boldsymbol{\beta}, \mathbf{x}) = \frac{1}{1 + e^{-\boldsymbol{\beta}\mathbf{x}}}$$

When $y = 0, p(y = 0|\boldsymbol{\beta}, \mathbf{x}) = 1 - p(y = 1|\boldsymbol{\beta}, \mathbf{x}) = 1 - h_{\boldsymbol{\beta}}(\mathbf{x})$

$$\boxed{p(y|\boldsymbol{\beta}, \mathbf{x}) = h_{\boldsymbol{\beta}}(\mathbf{x})^y (1 - h_{\boldsymbol{\beta}}(\mathbf{x}))^{1-y}}$$

$$\begin{aligned} \mathcal{L}(\boldsymbol{\beta}) = p(\mathbf{y}|\mathbf{X}, \boldsymbol{\beta}) &= \prod_{i=1}^{n} p(y|\mathbf{x}_i, \boldsymbol{\beta}) \\ &= \prod_{i=1}^{n} h_{\boldsymbol{\beta}}(\mathbf{x}_i)^{y_i} (1 - h_{\boldsymbol{\beta}}(\mathbf{x}_i))^{1-y_i} \end{aligned}$$

Logistic Regression

Given training data: p-dimensional features $\mathbf{x}_i$ with binary labels y_i for $i = 1, \ldots, n$ find the parameters $\boldsymbol{\beta} = (\beta_0, \beta_1, \ldots, \beta_p)$ that maximizes the likelihood $\mathcal{L}(\boldsymbol{\beta})$.

- Function to be maximized: $\mathcal{L}(\boldsymbol{\beta}) = \prod_{i=1}^{n} h_{\boldsymbol{\beta}}(\mathbf{x}_i)^{y_i}(1 - h_{\boldsymbol{\beta}}(\mathbf{x}_i))^{1-y_i}$
- Unknown parameters: $\boldsymbol{\beta} = (\beta_0, \beta_1, \ldots, \beta_p)$

Logistic Regression

Given training data: p-dimensional features $\mathbf{x}_i$ with binary labels y_i for $i = 1, \ldots, n$ find the parameters $\boldsymbol{\beta} = (\beta_0, \beta_1, \ldots, \beta_p)$ that maximizes the likelihood $\mathcal{L}(\boldsymbol{\beta})$.

- Function to be maximized: $\mathcal{L}(\boldsymbol{\beta}) = \prod_{i=1}^{n} h_{\boldsymbol{\beta}}(\mathbf{x}_i)^{y_i}(1 - h_{\boldsymbol{\beta}}(\mathbf{x}_i))^{1-y_i}$
- Unknown parameters: $\boldsymbol{\beta} = (\beta_0, \beta_1, \ldots, \beta_p)$

Gradient Descent: $\boxed{\boldsymbol{\beta}_{t+1} = \boldsymbol{\beta}_t + \epsilon \nabla \mathcal{L}(\boldsymbol{\beta}_t)}$

Logistic Regression

$\log \mathcal{L}(\boldsymbol{\beta})$ is a monotonic function of $\mathcal{L}(\boldsymbol{\beta}) \Rightarrow$ equivalent to maximixe the log-likelihood.

- Function to be maximized:
  $\log \mathcal{L}(\boldsymbol{\beta}) = \log \prod_{i=1}^{n} h_{\boldsymbol{\beta}}(\mathbf{x}_i)^{y_i}(1 - h_{\boldsymbol{\beta}}(\mathbf{x}_i))^{1-y_i}$
- Unknown parameters: $\boldsymbol{\beta} = (\beta_0, \beta_1, \ldots, \beta_p)$

Gradient Descent: $\boxed{\boldsymbol{\beta}_{t+1} = \boldsymbol{\beta}_t + \epsilon \nabla \log \mathcal{L}(\boldsymbol{\beta}_t)}$

Logistic Regression: Gradient of the Likelihood

$$\nabla \log \mathcal{L}(\boldsymbol{\beta}_t) \quad = \quad \nabla \Big(\log \big(\prod_{i=1}^{n} h_{\boldsymbol{\beta}}(\mathbf{x}_i)^{y_i} (1 - h_{\boldsymbol{\beta}}(\mathbf{x}_i))^{1-y_i} \big) \Big)$$

Logistic Regression: Gradient of the Likelihood

$$
\begin{aligned}
\nabla \log \mathcal{L}(\boldsymbol{\beta}_t) &= \nabla \Big(\log \big(\prod_{i=1}^{n} h_{\boldsymbol{\beta}}(\mathbf{x}_i)^{y_i} (1 - h_{\boldsymbol{\beta}}(\mathbf{x}_i))^{1-y_i} \big) \Big) \\
&= \nabla \sum_{i=1}^{n} \Big(\log h_{\boldsymbol{\beta}}(\mathbf{x}_i)^{y_i} + \log \big(1 - h_{\boldsymbol{\beta}}(\mathbf{x}_i)\big)^{1-y_i} \Big)
\end{aligned}
$$

Logistic Regression: Gradient of the Likelihood

$$
\begin{aligned}
\nabla \log \mathcal{L}(\boldsymbol{\beta}_t) &= \nabla \Big(\log \big(\prod_{i=1}^{n} h_{\boldsymbol{\beta}}(\mathbf{x}_i)^{y_i} (1 - h_{\boldsymbol{\beta}}(\mathbf{x}_i))^{1-y_i} \big) \Big) \\
&= \nabla \sum_{i=1}^{n} \Big(\log h_{\boldsymbol{\beta}}(\mathbf{x}_i)^{y_i} + \log \big(1 - h_{\boldsymbol{\beta}}(\mathbf{x}_i)\big)^{1-y_i} \Big) \\
&= \nabla \sum_{i=1}^{n} \Big(y_i \log h_{\boldsymbol{\beta}}(\mathbf{x}_i) + (1 - y_i) \log \big(1 - h_{\boldsymbol{\beta}}(\mathbf{x}_i)\big) \Big)
\end{aligned}
$$

Logistic Regression: Gradient of the Likelihood

$$
\begin{aligned}
\nabla \log \mathcal{L}(\boldsymbol{\beta}_t) &= \nabla \Big(\log \big(\prod_{i=1}^{n} h_{\boldsymbol{\beta}}(\mathbf{x}_i)^{y_i} (1 - h_{\boldsymbol{\beta}}(\mathbf{x}_i))^{1-y_i} \big) \Big) \\
&= \nabla \sum_{i=1}^{n} \Big(\log h_{\boldsymbol{\beta}}(\mathbf{x}_i)^{y_i} + \log \big(1 - h_{\boldsymbol{\beta}}(\mathbf{x}_i)\big)^{1-y_i} \Big) \\
&= \nabla \sum_{i=1}^{n} \Big(y_i \log h_{\boldsymbol{\beta}}(\mathbf{x}_i) + (1 - y_i) \log \big(1 - h_{\boldsymbol{\beta}}(\mathbf{x}_i)\big) \Big) \\
&= \sum_{i=1}^{n} \Big(y_i \nabla \log h_{\boldsymbol{\beta}}(\mathbf{x}_i) + (1 - y_i) \nabla \log \big(1 - h_{\boldsymbol{\beta}}(\mathbf{x}_i)\big) \Big)
\end{aligned}
$$

Logistic Regression: Gradient of the Likelihood

$$
\begin{aligned}
\nabla \log \mathcal{L}(\boldsymbol{\beta}_t) &= \nabla \Big(\log \big(\prod_{i=1}^{n} h_{\boldsymbol{\beta}}(\mathbf{x}_i)^{y_i} (1 - h_{\boldsymbol{\beta}}(\mathbf{x}_i))^{1-y_i} \big) \Big) \\
&= \nabla \sum_{i=1}^{n} \Big(\log h_{\boldsymbol{\beta}}(\mathbf{x}_i)^{y_i} + \log (1 - h_{\boldsymbol{\beta}}(\mathbf{x}_i))^{1-y_i} \Big) \\
&= \nabla \sum_{i=1}^{n} \big(y_i \log h_{\boldsymbol{\beta}}(\mathbf{x}_i) + (1 - y_i) \log (1 - h_{\boldsymbol{\beta}}(\mathbf{x}_i)) \big) \\
&= \sum_{i=1}^{n} \big(y_i \nabla \log h_{\boldsymbol{\beta}}(\mathbf{x}_i) + (1 - y_i) \nabla \log (1 - h_{\boldsymbol{\beta}}(\mathbf{x}_i)) \big) \\
&= \boxed{\sum_{i=1}^{n} \Big(\frac{y_i}{h_{\boldsymbol{\beta}}(\mathbf{x}_i)} \nabla h_{\boldsymbol{\beta}}(\mathbf{x}_i) - \frac{(1 - y_i)}{(1 - h_{\boldsymbol{\beta}}(\mathbf{x}_i))} \nabla h_{\boldsymbol{\beta}}(\mathbf{x}_i) \Big)}
\end{aligned}
$$

Logistic Regression: Gradient of the Likelihood

$$
\begin{aligned}
\nabla h_{\beta}(\mathbf{x}_i) &= \nabla\left(\frac{1}{1+e^{-\beta\mathbf{x}_i}}\right) \\
&= -1.\left(1+e^{-(\beta\mathbf{x}_i)}\right)^{-2}.e^{-(\beta\mathbf{x}_i)}.-\mathbf{x}_i \\
&= \mathbf{x}_i.\frac{1}{1+e^{-(\beta\mathbf{x}_i)}}.\frac{e^{-(\beta\mathbf{x}_i)}}{1+e^{-(\beta\mathbf{x}_i)}} \\
&= \mathbf{x}_i.\frac{1}{1+e^{-(\beta\mathbf{x}_i)}}.\left(1-\frac{1}{1+e^{-(\beta\mathbf{x}_i)}}\right) \\
&= \mathbf{x}_i\,.\,h_{\beta}(\mathbf{x}_i)\,.\,(1-h_{\beta}(\mathbf{x}_i))
\end{aligned}
$$

Logistic Regression: Gradient of the Likelihood

$$\nabla \log \mathcal{L}(\boldsymbol{\beta}_t) = \sum_{i=1}^{n} \Big(\frac{y_i}{h_{\boldsymbol{\beta}}(\mathbf{x}_i)} \nabla h_{\boldsymbol{\beta}}(\mathbf{x}_i) - \frac{(1-y_i)}{(1-h_{\boldsymbol{\beta}}(\mathbf{x}_i))} \nabla h_{\boldsymbol{\beta}}(\mathbf{x}_i) \Big)$$

Logistic Regression: Gradient of the Likelihood

$$\nabla \log \mathcal{L}(\boldsymbol{\beta}_t) = \sum_{i=1}^{n} \Big(\frac{y_i}{h_{\boldsymbol{\beta}}(\mathbf{x}_i)} \nabla h_{\boldsymbol{\beta}}(\mathbf{x}_i) - \frac{(1-y_i)}{(1-h_{\boldsymbol{\beta}}(\mathbf{x}_i))} \nabla h_{\boldsymbol{\beta}}(\mathbf{x}_i) \Big)$$

$$= \sum_{i=1}^{n} \Big(\frac{y_i}{h_{\boldsymbol{\beta}}(\mathbf{x}_i)} (\mathbf{x}_i h_{\boldsymbol{\beta}}(\mathbf{x}_i)(1-h_{\boldsymbol{\beta}}(\mathbf{x}_i))) - \frac{(1-y_i)}{(1-h_{\boldsymbol{\beta}}(\mathbf{x}_i))} (\mathbf{x}_i h_{\boldsymbol{\beta}}(\mathbf{x}_i)(1-h_{\boldsymbol{\beta}}(\mathbf{x}_i))) \Big)$$

Logistic Regression: Gradient of the Likelihood

$$\nabla \log \mathcal{L}(\boldsymbol{\beta}_t) = \sum_{i=1}^{n} \Big(\frac{y_i}{h_{\boldsymbol{\beta}}(\mathbf{x}_i)} \nabla h_{\boldsymbol{\beta}}(\mathbf{x}_i) - \frac{(1-y_i)}{(1-h_{\boldsymbol{\beta}}(\mathbf{x}_i))} \nabla h_{\boldsymbol{\beta}}(\mathbf{x}_i) \Big)$$

$$= \sum_{i=1}^{n} \Big(\frac{y_i}{h_{\boldsymbol{\beta}}(\mathbf{x}_i)} (\mathbf{x}_i h_{\boldsymbol{\beta}}(\mathbf{x}_i)(1-h_{\boldsymbol{\beta}}(\mathbf{x}_i))) - \frac{(1-y_i)}{(1-h_{\boldsymbol{\beta}}(\mathbf{x}_i))} (\mathbf{x}_i h_{\boldsymbol{\beta}}(\mathbf{x}_i)(1-h_{\boldsymbol{\beta}}(\mathbf{x}_i))) \Big)$$

$$= \sum_{i=1}^{n} \big(y_i \mathbf{x}_i (1-h_{\boldsymbol{\beta}}(\mathbf{x}_i)) - (1-y_i) \mathbf{x}_i h_{\boldsymbol{\beta}}(\mathbf{x}_i) \big)$$

Logistic Regression: Gradient of the Likelihood

$$\nabla \log \mathcal{L}(\boldsymbol{\beta}_t) = \sum_{i=1}^{n} \Big(\frac{y_i}{h_{\boldsymbol{\beta}}(\mathbf{x}_i)} \nabla h_{\boldsymbol{\beta}}(\mathbf{x}_i) - \frac{(1-y_i)}{(1-h_{\boldsymbol{\beta}}(\mathbf{x}_i))} \nabla h_{\boldsymbol{\beta}}(\mathbf{x}_i) \Big)$$

$$= \sum_{i=1}^{n} \Big(\frac{y_i}{h_{\boldsymbol{\beta}}(\mathbf{x}_i)} (\mathbf{x}_i h_{\boldsymbol{\beta}}(\mathbf{x}_i)(1-h_{\boldsymbol{\beta}}(\mathbf{x}_i))) - \frac{(1-y_i)}{(1-h_{\boldsymbol{\beta}}(\mathbf{x}_i))} (\mathbf{x}_i h_{\boldsymbol{\beta}}(\mathbf{x}_i)(1-h_{\boldsymbol{\beta}}(\mathbf{x}_i))) \Big)$$

$$= \sum_{i=1}^{n} \Big(y_i \mathbf{x}_i (1-h_{\boldsymbol{\beta}}(\mathbf{x}_i)) - (1-y_i) \mathbf{x}_i h_{\boldsymbol{\beta}}(\mathbf{x}_i) \Big)$$

$$= \sum_{i=1}^{n} \Big(y_i \mathbf{x}_i - y_i \mathbf{x}_i h_{\boldsymbol{\beta}}(\mathbf{x}_i) - \mathbf{x}_i h_{\boldsymbol{\beta}}(\mathbf{x}_i) + y_i \mathbf{x}_i h_{\boldsymbol{\beta}}(\mathbf{x}_i) \Big)$$

Logistic Regression: Gradient of the Likelihood

$$\nabla \log \mathcal{L}(\boldsymbol{\beta}_t) = \sum_{i=1}^{n} \Big(\frac{y_i}{h_{\boldsymbol{\beta}}(\mathbf{x}_i)} \nabla h_{\boldsymbol{\beta}}(\mathbf{x}_i) - \frac{(1-y_i)}{(1-h_{\boldsymbol{\beta}}(\mathbf{x}_i))} \nabla h_{\boldsymbol{\beta}}(\mathbf{x}_i) \Big)$$

$$= \sum_{i=1}^{n} \Big(\frac{y_i}{h_{\boldsymbol{\beta}}(\mathbf{x}_i)} (\mathbf{x}_i h_{\boldsymbol{\beta}}(\mathbf{x}_i)(1-h_{\boldsymbol{\beta}}(\mathbf{x}_i))) - \frac{(1-y_i)}{(1-h_{\boldsymbol{\beta}}(\mathbf{x}_i))} (\mathbf{x}_i h_{\boldsymbol{\beta}}(\mathbf{x}_i)(1-h_{\boldsymbol{\beta}}(\mathbf{x}_i))) \Big)$$

$$= \sum_{i=1}^{n} \Big(y_i \mathbf{x}_i (1-h_{\boldsymbol{\beta}}(\mathbf{x}_i)) - (1-y_i) \mathbf{x}_i h_{\boldsymbol{\beta}}(\mathbf{x}_i) \Big)$$

$$= \sum_{i=1}^{n} \Big(y_i \mathbf{x}_i - y_i \mathbf{x}_i h_{\boldsymbol{\beta}}(\mathbf{x}_i) - \mathbf{x}_i h_{\boldsymbol{\beta}}(\mathbf{x}_i) + y_i \mathbf{x}_i h_{\boldsymbol{\beta}}(\mathbf{x}_i) \Big)$$

$$= \sum_{i=1}^{n} \Big(y_i \mathbf{x}_i (1-h_{\boldsymbol{\beta}}(\mathbf{x}_i)) - (1-y_i) \mathbf{x}_i h_{\boldsymbol{\beta}}(\mathbf{x}_i) \Big)$$

Logistic Regression: Gradient of the Likelihood

$$\nabla \log \mathcal{L}(\boldsymbol{\beta}_t) = \sum_{i=1}^{n} \Big(\frac{y_i}{h_{\boldsymbol{\beta}}(\mathbf{x}_i)} \nabla h_{\boldsymbol{\beta}}(\mathbf{x}_i) - \frac{(1-y_i)}{(1-h_{\boldsymbol{\beta}}(\mathbf{x}_i))} \nabla h_{\boldsymbol{\beta}}(\mathbf{x}_i) \Big)$$

$$= \sum_{i=1}^{n} \Big(\frac{y_i}{h_{\boldsymbol{\beta}}(\mathbf{x}_i)} (\mathbf{x}_i h_{\boldsymbol{\beta}}(\mathbf{x}_i)(1-h_{\boldsymbol{\beta}}(\mathbf{x}_i))) - \frac{(1-y_i)}{(1-h_{\boldsymbol{\beta}}(\mathbf{x}_i))} (\mathbf{x}_i h_{\boldsymbol{\beta}}(\mathbf{x}_i)(1-h_{\boldsymbol{\beta}}(\mathbf{x}_i))) \Big)$$

$$= \sum_{i=1}^{n} \Big(y_i \mathbf{x}_i (1-h_{\boldsymbol{\beta}}(\mathbf{x}_i)) - (1-y_i) \mathbf{x}_i h_{\boldsymbol{\beta}}(\mathbf{x}_i) \Big)$$

$$= \sum_{i=1}^{n} \Big(y_i \mathbf{x}_i - y_i \mathbf{x}_i h_{\boldsymbol{\beta}}(\mathbf{x}_i) - \mathbf{x}_i h_{\boldsymbol{\beta}}(\mathbf{x}_i) + y_i \mathbf{x}_i h_{\boldsymbol{\beta}}(\mathbf{x}_i) \Big)$$

$$= \sum_{i=1}^{n} \Big(y_i \mathbf{x}_i (1-h_{\boldsymbol{\beta}}(\mathbf{x}_i)) - (1-y_i) \mathbf{x}_i h_{\boldsymbol{\beta}}(\mathbf{x}_i) \Big)$$

$$= \sum_{i=1}^{n} \Big(\mathbf{x}_i (y_i - h_{\boldsymbol{\beta}}(\mathbf{x}_i)) \Big)$$

Logistic Regression: Gradient Descent

Gradient is a $p+1$ dimensional vector:

$$\left(\frac{\partial}{\partial \beta_0}\log \mathcal{L}(\boldsymbol{\beta})\ \frac{\partial}{\partial \beta_1}\log \mathcal{L}(\boldsymbol{\beta}) \ldots \frac{\partial}{\partial \beta_p}\log \mathcal{L}(\boldsymbol{\beta})\right)$$

In the j^{th} dimension, for $j > 0$:

$$\frac{\partial}{\partial \beta_j}\log \mathcal{L}(\boldsymbol{\beta}) = \sum_{i=1}^{n} x_{ij}(y_i - h_{\boldsymbol{\beta}}(\mathbf{x}_i))$$

For $j = 0$, $\mathbf{x}_i = \mathbf{1}$,

$$\frac{\partial}{\partial \beta_0}\log \mathcal{L}(\boldsymbol{\beta}) = \sum_{i=1}^{n} (y_i - h_{\boldsymbol{\beta}}(\mathbf{x}_i))$$

Logistic Regression: Gradient Descent

Gradient Descent procedure to estimate Logistic Regression parameters:

- Initialize $\beta^{(t)}$ for iteration $t = 0$
- Repeat until convergence: $\beta_j^{(t+1)} = \beta_j^{(t)} + \epsilon \sum_{i=1}^{n} x_{ij}(y_i - h_\beta(\mathbf{x}_i))$.

Gradient Descent for Neural Networks

Predicted output $\mathbf{y}$ with input $\mathbf{x}$ using network is $\mathbf{y} = f(\mathbf{x}, \boldsymbol{\theta})$, where network parameters (weights) are $\boldsymbol{\theta}$ and per-example Loss used is L.

- Initialize $\boldsymbol{\theta}^{(t)}$ for iteration $t = 0$
- Repeat until convergence:
  - Compute gradient estimate: $\hat{g} \leftarrow \nabla_{\boldsymbol{\theta}} \Sigma_i^n L(f(\mathbf{x}_i, \boldsymbol{\theta}^{(t)}), \mathbf{y}_i^{(t)})$
  - Update: $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \epsilon \hat{g}$.

ϵ: step size or learning rate

Superscript (t): denotes t^{th} iteration (not exponent)

Extensions and Variants of Gradient Descent

Stochastic Gradient Descent (SGD)

In Gradient Descent, we repeat until convergence:
$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \epsilon \nabla_{\boldsymbol{\theta}} \Sigma_i^n L(f(\mathbf{x}_i, \boldsymbol{\theta}^{(t)}), \mathbf{y}_i^{(t)})$.

Summation over n examples in each iteration: expensive!

SGD: noisy approximation to the true gradient but much faster.

- Iterate through the training data in *epochs*
- Compute gradient only at one/some (mini-batch) training points
- No guarantee of loss reducing at each step

Confusing Terms

Epoch: one pass over entire data (n examples)
Step: one update of the parameters (θ)
m: mini-batch size

Algorithm	Steps per Epoch
Gradient Descent	1
Mini-batch Gradient Descent	n/m
SGD	n

Stochastic Gradient Descent

- Initialize $\boldsymbol{\theta}^{(t)}$ for iteration $t = 0$
- Repeat until convergence:
  - Sample a minibatch of m examples $\{(\mathbf{x}_1, \mathbf{y}_1), \ldots (\mathbf{x}_m, \mathbf{y}_m)\}$
  - Compute gradient estimate: $\hat{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \Sigma_i^m L(f(\mathbf{x}_i, \boldsymbol{\theta}^{(t)}), \mathbf{y}_i^{(t)})$
  - Update: $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \epsilon \hat{g}$

Momentum

Momentum

Gradient descent follows the steepest path downwards; the progress is slow, steady. SGD does not use curvature information.

It has trouble navigating ravines (areas where the surface curves much more steeply in one dimension than in another).

So, SGD oscillates across the slopes of the ravine, making slow progress.

Image 2: SGD without momentum

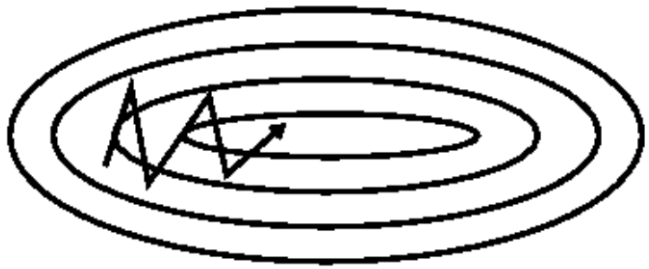


Image 3: SGD with momentum

Figure: `http://ruder.io/optimizing-gradient-descent/index.html`

Momentum

Momentum accumulates an exponentially decaying moving average of past gradients.

$$\mathbf{v}^{(t+1)} \leftarrow \alpha \mathbf{v}^{(t)} - \epsilon \hat{g}$$

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} + \mathbf{v}^{(t+1)}$$

The ball accumulates momentum as it rolls downhill, becoming faster and faster on the way.

Large α (relative to ϵ), more previous gradients affect current direction. Common values: 0.5, 0.9, 0.99.

Step size is large when many successive gradients point in the same direction.

Momentum

The momentum term increases for dimensions whose gradients point in the same directions.

$$\mathbf{v}^{(t+1)} \leftarrow \alpha \mathbf{v}^{(t)} - \epsilon \hat{g}$$

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} + \mathbf{v}^{(t+1)}$$

It reduces updates for dimensions whose gradients change directions.

As a result, we gain:

- faster convergence
- reduced oscillation

SGD with Momentum

- Initialize $\boldsymbol{\theta}^{(t)}, \mathbf{v}^{(t)}$ for iteration $t = 0$
- Repeat until convergence:
  - Sample a minibatch of m examples $\{(\mathbf{x}_1, \mathbf{y}_1), \ldots (\mathbf{x}_m, \mathbf{y}_m)\}$
  - Compute gradient estimate: $\hat{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \Sigma_i^m L(f(\mathbf{x}_i, \boldsymbol{\theta}^{(t)}), \mathbf{y}_i^{(t)})$
  - Compute velocity: $\mathbf{v}^{(t+1)} \leftarrow \alpha \mathbf{v}^{(t)} - \epsilon \hat{g}$
  - Update: $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} + \mathbf{v}^{(t+1)}$.

Momentum

Momentum: adding extra mass/inertia to the rolling ball: smoother, accelerator, dampens oscillations.

Momentum: short-term memory; repeatedly moving in the same direction, so can take larger steps.

Allows us to navigate through narrow valleys, small humps and local minima.

`https://distill.pub/2017/momentum/`

Learning Rate

Significantly affects model performance.

Among the most difficult hyperparameters to set: cost is highly sensitive to some directions in parameter space, insensitive to others.

Momentum: mitigates problems somewhat, but adds another hyperparameter. In regions with steep slopes, the already large gradient blows up further.

Heuristics: Tune with different rates - check for few epochs. Not always effective.

Algorithms with Adaptive Learning Rates

AdaGrad

Use different learning rate for each parameter.

Adapt the learning rates by scaling them inversely proportional to square root of sum of all historical squared value of gradient.

Parameters with largest gradient have rapid decrease in learning rate, parameters with small gradients have relatively small decrease in learning rate.

Empirically, found to work well with sparse data.

AdaGrad: Per-parameter Update

SGD Parameter Update: $\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \epsilon \hat{g}$

Accumulate the squared gradient (per-parameter): $\mathbf{r}^{(t+1)} \leftarrow \mathbf{r}^{(t)} + \hat{g}.\hat{g}$

AdaGrad Parameter Update: $\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \frac{\epsilon}{\sqrt{\delta + \mathbf{r}^{(t+1)}}}.\hat{g}$

δ: avoids division by zero, without square root, performance is worse!

Can store $\mathbf{r}$ in a matrix and vectorize parameter update through a matrix-vector product $\odot$

AdaGrad

- Initialize $\boldsymbol{\theta}^{(t)}, \mathbf{r}^{(t)} = 0$ for iteration $t = 0$
- Repeat until convergence:
  - Sample a minibatch of m examples $\{(\mathbf{x}_1, \mathbf{y}_1), \ldots (\mathbf{x}_m, \mathbf{y}_m)\}$
  - Compute gradient estimate: $\hat{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \Sigma_i^m L(f(\mathbf{x}_i, \boldsymbol{\theta}^{(t)}), \mathbf{y}_i^{(t)})$
  - Accumulate squared gradient per parameter: $\mathbf{r}^{(t+1)} \leftarrow \mathbf{r}^{(t)} + \hat{\mathbf{g}} \odot \hat{\mathbf{g}}$
  - Compute Update: $\Delta \leftarrow -\frac{\epsilon}{\sqrt{\delta + \mathbf{r}^{(t+1)}}} \odot \hat{\mathbf{g}}$
  - Apply Update: $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} + \Delta$

AdaGrad

Benefit: eliminates the need to manually tune the learning rate. Commonly initialized with 0.01.

Weakness: The accumulated sum of squared gradients keeps growing during training. This causes the learning rate to shrink and eventually become infinitesimally small, prematurely.

RMSProp

Designed to address the weakness of AdaGrad

By changing the gradient accumulation to an exponentially weighted moving average

Discards history from distant past

RMSProp

- Initialize $\boldsymbol{\theta}^{(t)}, \mathbf{r}^{(t)} = 0$ for iteration $t = 0$
- Repeat until convergence:
  - Sample a minibatch of m examples $\{(\mathbf{x}_1, \mathbf{y}_1), \ldots (\mathbf{x}_m, \mathbf{y}_m)\}$
  - Compute gradient estimate: $\hat{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \Sigma_i^m L(f(\mathbf{x}_i, \boldsymbol{\theta}^{(t)}), \mathbf{y}_i^{(t)})$
  - Accumulate squared gradient: $\mathbf{r}^{(t+1)} \leftarrow \rho \mathbf{r}^{(t)} + (1-\rho)\hat{\mathbf{g}} \odot \hat{\mathbf{g}}$
  - Compute Update: $\Delta \leftarrow -\frac{\epsilon}{\sqrt{\delta + \mathbf{r}^{(t+1)}}} \odot \hat{\mathbf{g}}$
  - Apply Update: $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} + \Delta$

RMSProp with Momentum

- Initialize $\boldsymbol{\theta}^{(t)}, \mathbf{v}^{(t)}, \mathbf{r}^{(t)} = 0$ for iteration $t = 0$
- Repeat until convergence:
  - Sample a minibatch of m examples $\{(\mathbf{x}_1, \mathbf{y}_1), \ldots (\mathbf{x}_m, \mathbf{y}_m)\}$
  - Compute gradient estimate: $\hat{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \Sigma_i^m L(f(\mathbf{x}_i, \boldsymbol{\theta}^{(t)}), \mathbf{y}_i^{(t)})$
  - Accumulate squared gradient: $\mathbf{r}^{(t+1)} \leftarrow \rho \mathbf{r}^{(t)} + (1-\rho)\hat{\mathbf{g}} \odot \hat{\mathbf{g}}$
  - Compute Velocity Update: $\mathbf{v}^{(t+1)} \leftarrow \alpha \mathbf{v}^{(t)} - \frac{\epsilon}{\sqrt{\delta + \mathbf{r}^{(t+1)}}} \odot \hat{\mathbf{g}}$
  - Apply Update: $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} + \mathbf{v}^{(t+1)}$

Adam

Adam: Adaptive Moment Estimation

Adam computes adaptive learning rates for each parameter.

Stores an exponentially decaying average of past squared gradients like RMSprop.

Additionally, Adam also keeps an exponentially decaying average of past gradients, similar to momentum.

Whereas momentum can be seen as a ball running down a slope, Adam behaves like a heavy ball with friction, which thus prefers flat minima in the error surface

Adam

Both Per-parameter Gradients and Squared Gradients are accumulated and used in Updates:

Accumulate gradient: $\mathbf{s}^{(t+1)} \leftarrow \frac{1}{1-\rho_1^t}(\rho_1 \mathbf{s}^{(t)} + (1-\rho_1)\hat{\mathbf{g}})$

Accumulate squared gradient: $\mathbf{r}^{(t+1)} \leftarrow \frac{1}{1-\rho_2^t}(\rho_2 \mathbf{r}^{(t)} + (1-\rho_2)\hat{\mathbf{g}} \odot \hat{\mathbf{g}})$

s and **r** are estimates of first moment (mean) and second moment (variance) of the gradients respectively.

These estimates are biased to zero esp during initial iterations and when ρ_1, ρ_2 are small. So, they are rescaled.

Adam is more or less the default choice now ($\rho_1 = 0.9, \rho = 0.999, \delta = 1e-8$).

Adam

- Initialize $\boldsymbol{\theta}^{(t)}, \mathbf{s}^{(t)} = 0, \mathbf{r}^{(t)} = 0$ for iteration $t = 0$
- Repeat until convergence:
  - Sample a minibatch of m examples $\{(\mathbf{x}_1, \mathbf{y}_1), \ldots (\mathbf{x}_m, \mathbf{y}_m)\}$
  - Compute gradient estimate: $\hat{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \Sigma_i^m L(f(\mathbf{x}_i, \boldsymbol{\theta}^{(t)}), \mathbf{y}_i^{(t)})$
  - Accumulate gradient: $\mathbf{s}^{(t+1)} \leftarrow \frac{1}{1-\rho_1^t}(\rho_1 \mathbf{s}^{(t)} + (1-\rho_1)\hat{\mathbf{g}})$
  - Accumulate squared gradient: $\mathbf{r}^{(t+1)} \leftarrow \frac{1}{1-\rho_2^t}(\rho_2 \mathbf{r}^{(t)} + (1-\rho_2)\hat{\mathbf{g}} \odot \hat{\mathbf{g}})$
  - Compute Update: $\Delta \leftarrow -\epsilon \frac{\mathbf{s}^{(t+1)}}{\delta + \sqrt{\mathbf{r}^{(t+1)}}}$
  - Apply Update: $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} + \Delta$

Second Order Methods

Optimization algorithms that use only the gradient, such as gradient descent, are called first-order optimization algorithms.

Optimization algorithms that also use the Hessian matrix, such as Newton's method, are called second-order optimization algorithms.

Newton's Algorithm

Principled way of finding the step size, using the Hessian.

- Initialize $\mathbf{x}^{(t)}$ for iteration $t = 0$
- Repeat until convergence: $\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \mathbf{H}^{-1}(\mathbf{x}^{(t)})\nabla f(\mathbf{x}^{(t)})$.

Computationally intensive: Hessian calculation in every iteration.

Hessian must be nonsingular (inverse must exist).

Other Methods

Conjugate Gradients
Broyden-Fletcher-Goldfarb-Shanno (BFGS)
Block Coordinate Descent
Continuation Methods
...

Which Optimizer?

Sparse data: adaptive learning rate helps. Adam is usually the default choice.

SGD often works well, may take longer. Relies heavily on initialization.

Improving Training and Generalization

Regularization

Optimization framework can be used to control model complexity and overfitting.

Instead of just maximizing $\mathcal{L}(\boldsymbol{\theta})$, we can maximize $\mathcal{L}(\boldsymbol{\theta}) - \lambda R(\boldsymbol{\theta})$

$R(\boldsymbol{\theta})$: function of $\boldsymbol{\theta}$
λ : hyperparameter that controls strength of regularization.

Norms

Norm: a function that assigns a length/size to each vector (that satisfies some properties)

Notion of length for n-dimensional vectors $\mathbf{x}$

- L1 or Absolute Value Norm: $||\mathbf{x}||_1 = \sum_i |x_i|$
- L2 or Euclidean Norm: $||\mathbf{x}||_2 = \sqrt{\sum_i x_i^2}$
- P-Norm: $||\mathbf{x}||_p = (\sum_i x_i^p)^{1/p}$

Logistic Regression

With L2-norm ($||\beta||_2$), gradient descent updates for logistic regression become:

Repeat until convergence: $\beta^{(t+1)} = \beta^t + \epsilon \sum_{i=1}^{n} \mathbf{x}_i(y_i - h_\beta(\mathbf{x}_i)) - \lambda\beta$.

Parameter Initialization

Affects both optimization and generalization

No theroretical understanding: mainly empirical evidence

Initial parameters need to ‘break symmetry’ between different units: two hidden units with same activation function are connected to the same inputs then these units must have different initial parameters

Parameter Initialization

Common strategy: Values drawn randomly from Gaussian or Uniform distribution

Choice of distribution affects less than scale (variance)

Large initial weights: stronger symmetry-breaking effect

Opposing perspectives: regularization and optimization

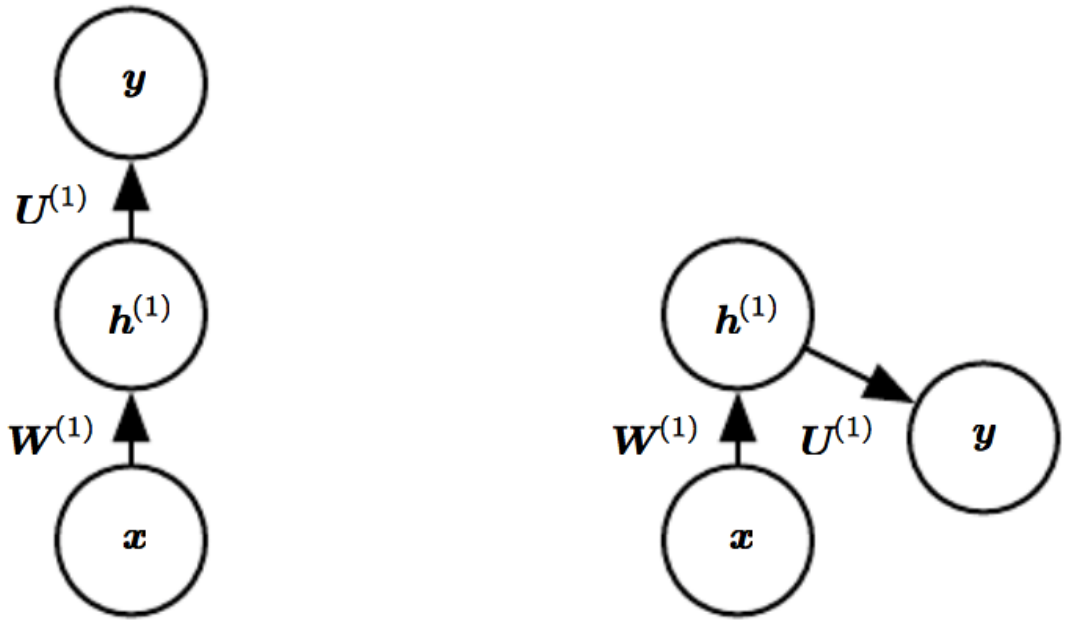
Initial scale of weights: hyperparameter

Pre-Training

Helps in both optimization and generalization

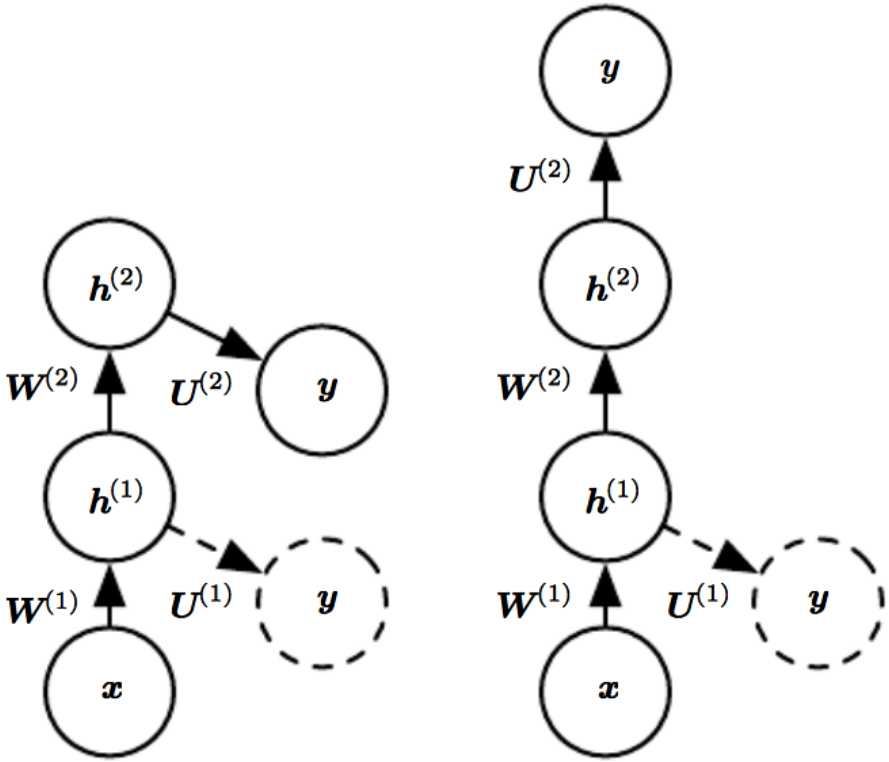
Several different ways of pretraining - depending on the architecture

Greedy Layerwise Pretraining



Start by training a shallow architecture.

Greedy Layerwise Pretraining



Retain the first layer and discard the hidden-to-output layer.
Add another pretrained layer.
Jointly fine-tune either at each stage or at the end.

Other Strategies to Improve SGD Performance

- Shuffle training data after each epoch.
- Batch normalization: normalize each mini-batch and backpropagate through the operations. Also acts as a regularizer.
- Early Stopping for regularization.
- Gradient Noise: add white noise to each gradient update.

More details: Lecun et al., "Efficient BackProp"

`http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf`

Hyperparameters

Number of hidden layers
Number of neurons in the hidden layers
Activation function

Learning Rate
Batch size
Convergence threshold
Regularization strength
...

Hyperparameter Tuning

Manual tuning infeasible for large, complex models

Popular Options: Random Search, Grid Search

Too time-consuming, not very effective

Hyperparameter Tuning

Can also be framed as an optimization problem:

$$\mathbf{p}^* = \arg\min_{\mathbf{p} \in \mathcal{P}} L(\mathbf{p})$$

where $\mathbf{p}^*$ denotes the optimal values for all B hyperparameters and the objective function is the network's loss function.

Note that model training optimizes L with respect to the model parameters, and not the hyperparameters.

The functional form of $L(\mathbf{p})$ is not known but its value, for any given input $\mathbf{p}$, can be computed: Black-box function.

Black-box function optimization problem: Bayesian Optimization

Bayesian Optimization

Bayesian Optimization (BO): sequential model-based approach for solving the black-box function optimization problem

Key idea: learn a *surrogate model* $\mathcal{M}$ that captures our beliefs about the unknown objective function ($L(\mathbf{p})$)

This model is learnt from *data*, $\mathcal{D}_n = (\mathbf{p}_1, L_1), \ldots, (\mathbf{p}_n, L_n)$, that consists of sequential evaluations of $L(\mathbf{p})$ for different values of $\mathbf{p}$

Bayesian Optimization

Generating the data sequence $\mathcal{D}_n = (\mathbf{p}_1, L_1), \ldots, (\mathbf{p}_n, L_n)$ requires making the decision of which $\mathbf{p}$ to evaluate next, at each step

This decision is made through an *acquisition function*

These functions are designed to have optima at points with

- high uncertainty in the surrogate model [exploration] and/or
- high predictive values in the surrogate model [exploitation]

Acquisition functions have known functional forms and are usually easier to optimize than the original objective function

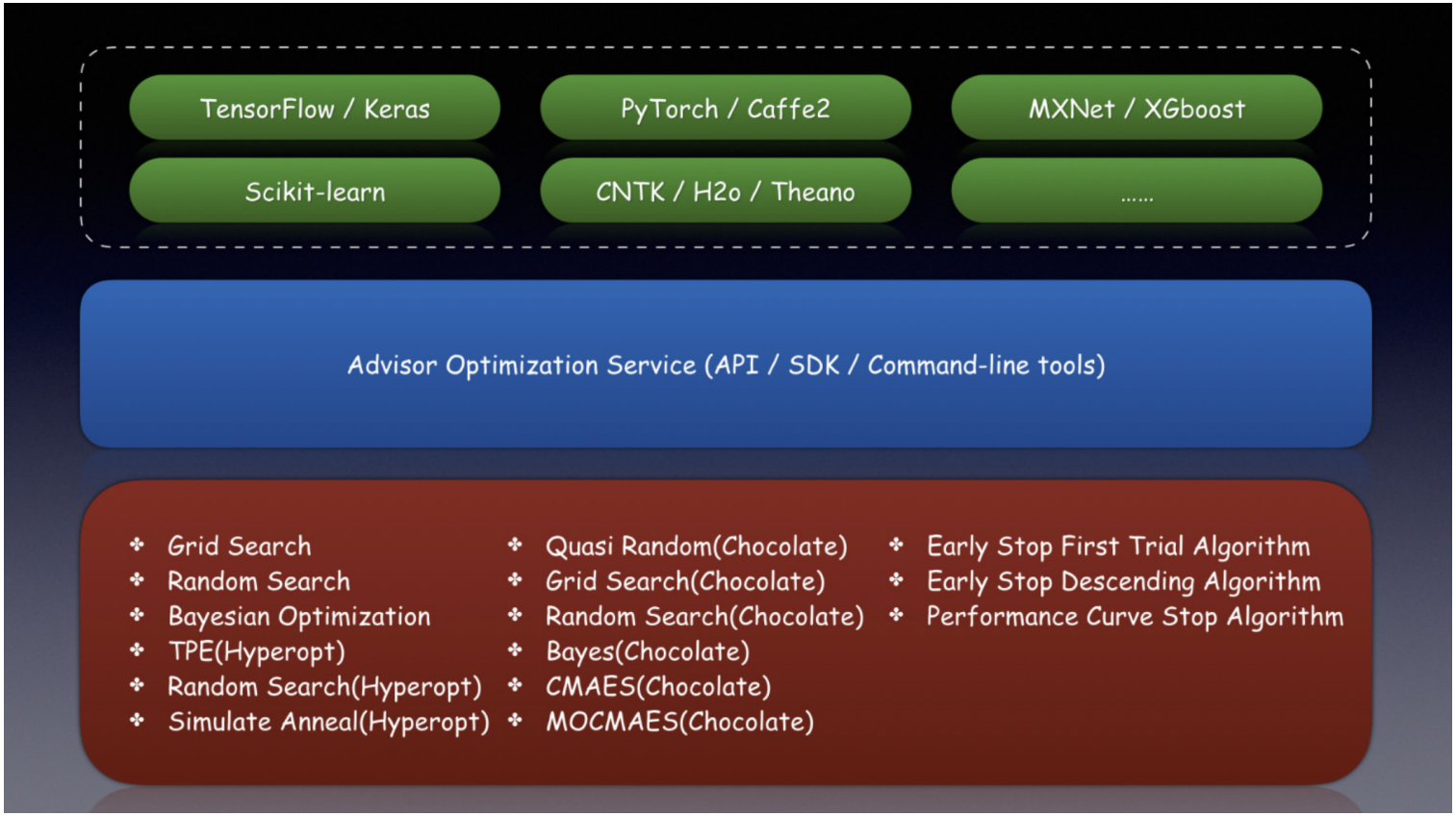
Bayesian Optimization

- For $n = 1, 2, , \ldots$
  - Select $\mathbf{p}_{n+1}$ by optimizing the acquisition function
  - Run network with hyperparameters $\mathbf{p}_{n+1}$ and obtain loss function value
  - Augment data $\mathcal{D}_n$ with loss function value and update surrogate model $\mathcal{M}$

Shahriari et al., "Taking the Human Out of the Loop: A Review of Bayesian Optimization"

`https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7352306`

Google Vizier



https://github.com/tobegit3hub/advisor

Concluding Remarks

Summary

Gradient Descent Algorithms

- Gradient
- Momentum
- Adaptive Learning Rates

Other algorithms use more information (e.g. curvature) or advanced heuristics but are computationally expensive

Several strategies to improve optimization and generalization of SGD-based algorithms

Design Models to Aid Optimization

It is more important to choose a model family that is easy to optimize than to use a powerful optimization algorithm.

Most recent advances have been obtained by careful modeling. Decades old SGD and momentum are still used in state-of-the-art networks.

Design Models to Aid Optimization

Model Design Strategies:

- Linear transformations between layers (e.g. ReLu, maxout)
- Differentiable activation functions
- Linear paths/skip connections to reduce length of shortest path between deep layers to mitigate vanishing gradient problem
- Extra copies of output attached to intermediate hidden layers of the network (e.g. GoogleLeNet): lower layers receive a large gradient.

References

- `http://www.deeplearningbook.org` (Chapters 4,8)
  - All figures in this presentation from here, unless stated otherwise
- Joel Grus, Data Science from Scratch (Chapter 8)
- `http://ruder.io/optimizing-gradient-descent/index.html`
- `https://www.cse.iitm.ac.in/~miteshk/CS7015.html`