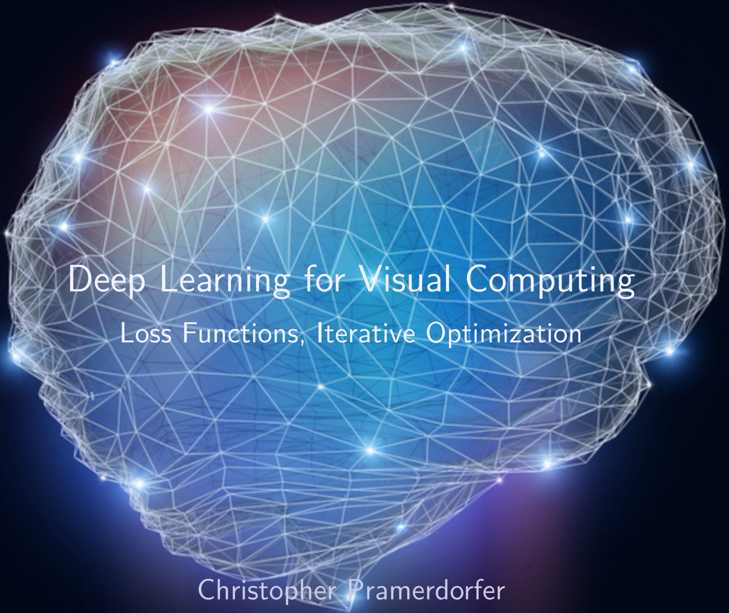


Deep Learning for Visual Computing

Loss Functions, Iterative Optimization

Christopher Pramerdorfer

Computer Vision Lab, TU Wien

Image by Oracle

Topics

How good is our classifier?

- Cross-entropy loss

How should we adapt the parameters?

- Iterative optimization
- Gradient Descent

Training Parametric Models

In parametric models f depends on parameters $\boldsymbol{\theta}$

- We write $\mathbf{w} = f(\mathbf{x}; \boldsymbol{\theta})$
- Training entails finding good parameters

For training any parametric model we need

- A loss function
- An optimization algorithm

Loss Functions

A loss function $L(\boldsymbol{\theta})$ (or cost or objective function)

- Measures performance of $f(\cdot\,; \boldsymbol{\theta})$ (lower loss is better)
- On some (training) dataset $\mathcal{D} = \{(\mathbf{x}_s, \mathbf{w}_s)\}_{s=1}^{S}$
- With respect to parameters $\boldsymbol{\theta}$

Choice of L depends on task

- Most popular classification loss is cross-entropy

Loss Functions
Cross-Entropy

Given two probability mass functions $\mathbf{u}$ and $\mathbf{v}$ in $\mathbb{R}^T$

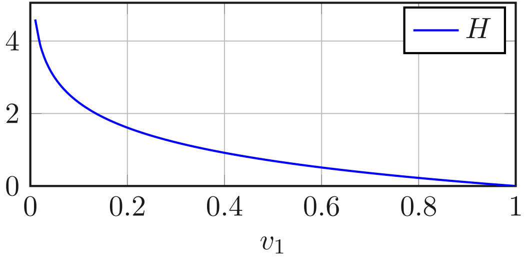
- $\mathbf{u} = (u_1, \ldots, u_T)$ and $\mathbf{v} = (v_1, \ldots, v_T)$

The cross-entropy between $\mathbf{u}$ and $\mathbf{v}$ is

$$H(\mathbf{u}, \mathbf{v}) = -\sum_{t=1}^{T} u_t \ln v_t$$

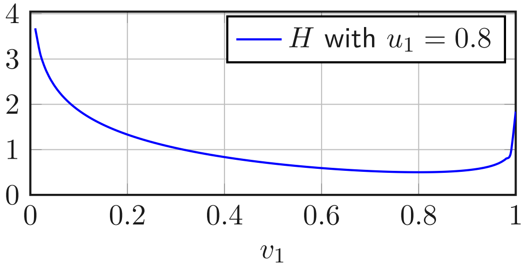
Example with $T = 2$ and $u_1 = 1$

- The more different $\mathbf{u}$ and $\mathbf{v}$ the higher H
- H measures the dissimilarity between $\mathbf{u}$ and $\mathbf{v}$



Note that H can reach 0 only if any $u_t = 1$

- In general $H(\mathbf{u}, \mathbf{v})$ = entropy of $\mathbf{u}$ if $\mathbf{u} = \mathbf{v}$



To utilize the cross-entropy for classifier training we

- Let $\mathbf{u}$ encode the ground-truth label, $u_c = 1$
- Let $\mathbf{v}$ be the predicted softmax class scores

H measures how dissimilar true and predicted probabilities are

- How well the classifier performs on a single sample

On this basis we calculate the cross-entropy loss on $\mathcal{D}$ as

$$L(\boldsymbol{\theta}) = \frac{1}{S}\sum_{s=1}^{S} H(\mathbf{w}_s, \mathsf{softmax}(f(\mathbf{x}_s; \boldsymbol{\theta})))$$

Average cross-entropy over some dataset $\mathcal{D}$

- We will use (subsets of) the training set as $\mathcal{D}$

Models trained with this loss are called softmax classifiers

- Also called logistic regression if $T = 2$

Classifiers learn to predict probabilities per class label

- In theory predictions are reliable probability estimates
- In practice DL classifiers are often overconfident

Gradient Descent
Motivation

We now know how to compute $L(\boldsymbol{\theta})$ for classification

Need a way to minimize $L(\boldsymbol{\theta})$

- Maximizes the training set classification performance
- And hopefully also validation/test performance (more later)

$L(\boldsymbol{\theta})$ is not linear in $\boldsymbol{\theta}$

- Need a nonlinear optimization algorithm
- Gradient Descent is popular choice in Deep Learning (DL)

Gradient Descent
Introduction

Assume terrain corresponds to $L(\boldsymbol{\theta})$ with $\dim(\boldsymbol{\theta}) = 2$

How do I get from location $\boldsymbol{\theta}$ to location of minimum $\hat{\boldsymbol{\theta}}$?

Gradient Descent
Introduction

Without actually seeing $L(\boldsymbol{\theta})$?

Gradient Descent
Introduction

Feel slope with feet, step in direction that feels steepest

- Again and again until ground feels flat

Gradient Descent
Definition

Iterative Optimization algorithm

In every iteration we

- Compute gradient $\boldsymbol{\theta}' = \nabla L(\boldsymbol{\theta})$
- Update parameters $\boldsymbol{\theta} = \boldsymbol{\theta} - \alpha\boldsymbol{\theta}'$

Hyperparameter $\alpha > 0$ is called learning rate

- Final step size is $\alpha \left\|\boldsymbol{\theta}'\right\|$

Let $f(x_1, \ldots, x_n)$ be a differentiable, real-valued function

The partial derivative f_{x_i} of f with respect to x_i

- Is also a real-valued function $f_{x_i}(x_1, \ldots, x_n)$

$f_{x_i}(\mathbf{x})$ encodes

- How fast f changes with argument x_i
- At some location $\mathbf{x}$

Gradient ∇f is vector of all partial derivatives of f

- $\nabla f = (f_{x_1}, \dots, f_{x_n})$
- Vector-valued function $\mathbb{R}^n \mapsto \mathbb{R}^n$

$\nabla f(\mathbf{x}) = (f_{x_1}(\mathbf{x}), \dots, f_{x_n}(\mathbf{x}))$ encodes

- How fast f changes with all arguments $x_1 \cdots x_n$
- At some location $\mathbf{x}$

Gradient Descent
Gradients

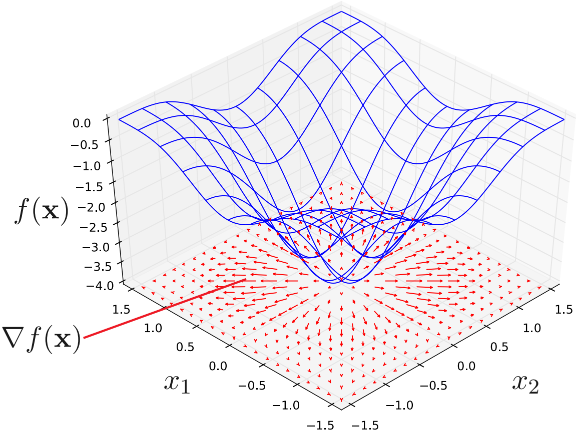


Image adapted from wikimedia.org

$\nabla f(\mathbf{x})$ specifies how f changes locally at $\mathbf{x}$

- Points in direction of greatest increase
- Norm equals magnitude of increase

Exactly what we need to minimize L

- Compute direction of greatest increase $\nabla L(\boldsymbol{\theta})$
- Move in the opposite direction

We stop if $\nabla L(\boldsymbol{\theta}) \approx \mathbf{0}$ (if norm is close to 0)

- No information where to go next
- L is flat at current location
- The case if we are at $\hat{\boldsymbol{\theta}}$ (but not only then)

Gradient Descent
Remarks

Simple and general algorithm

- Requires only that f is differentiable, real-valued
- Efficient (requires only first derivatives)

Several (possible) limitations

- Performs poorly for many f
- But works remarkable well with DL models

Algorithm stops if $\nabla L(\boldsymbol{\theta}) \approx \mathbf{0}$

- Applies to all critical points, not only minimum
- Should stop only at minimum

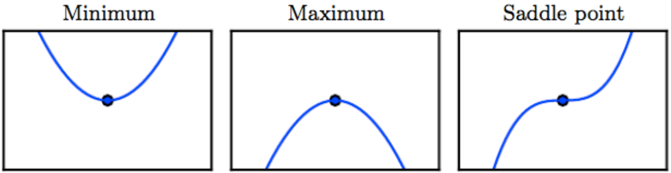


Image from [1]

Gradient Descent
Limitations – Critical Points and Local Minima

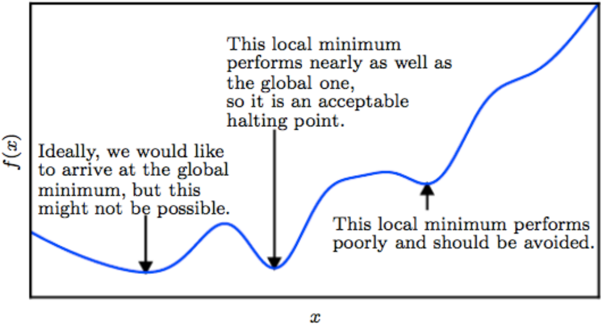


Image from [1]

Gradient Descent
Limitations – Critical Points and Local Minima

Algorithm stops at first minimum as $\nabla L(\boldsymbol{\theta}) \approx \mathbf{0}$

- But L generally has several local minima
- Algorithm usually finds only a local minimum

For loss functions of DL models evaluated on minibatches (below)

- Local minima are usually close to global minimum
- Optimization does not come close to critical points

Very different curvature in different directions (canyon-like)

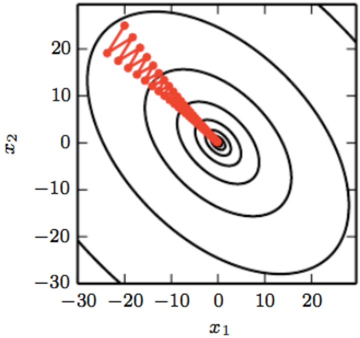


Image from [1]

Gradient Descent
Limitations – Poorly Conditioned Hessian

Gradient descent wastes time jumping between canyon walls

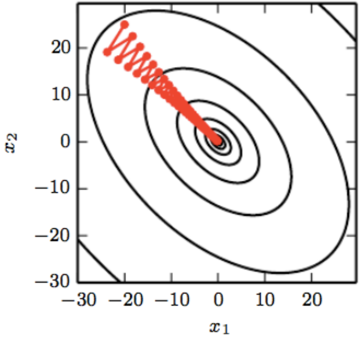


Image from [1]

Momentum improves speed of convergence by

- Dampening oscillations (previous slide)
- Increasing step size dynamically

Use exponential moving average of gradients for direction $\mathbf{v}$

- Influence of older gradients decays exponentially

Gradient Descent
Momentum

Iteration of gradient descent with momentum

- Update velocity $\mathbf{v} = \beta\mathbf{v} - \alpha\nabla L(\boldsymbol{\theta})$
- Update parameters $\boldsymbol{\theta} = \boldsymbol{\theta} + \mathbf{v}$

Hyperparameter $\beta \in [0, 1)$ called momentum

- Defines decay speed and maximum step size

Gradient Descent
Momentum

$\mathbf{v}$ builds up momentum if successive gradients are similar

- Improves speed of convergence

Maximum step size is $\alpha\|\mathbf{g}\|/(1-\beta)$

- Assuming the gradient is always $\mathbf{g}$
- At $\beta = 0.9$ maximum increase by factor of 10

Red is path, black are steepest descent directions

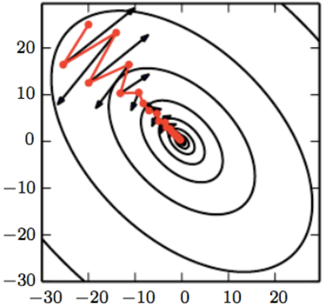


Image from [1]

Evaluate gradient at $\boldsymbol{\theta} + \mathbf{v}$ instead of $\boldsymbol{\theta}$

Iteration of gradient descent with Nesterov momentum

- Update velocity $\mathbf{v} = \beta\mathbf{v} - \alpha\nabla L(\boldsymbol{\theta} + \mathbf{v})$
- Update parameters $\boldsymbol{\theta} = \boldsymbol{\theta} + \mathbf{v}$

Often works better than standard momentum

Goal is to minimize $L(\boldsymbol{\theta})$ as measured on training data

Obvious choice is to use whole training set

- Called Batch Gradient Descent
- Time complexity per iteration increases linearly with S
- Problematic if S is large (need many iterations)

To solve this problem we

- Process the whole training set
- In minibatches of size S (one per iteration)

Possible because gradient is an expectation

- Can estimate training set loss on subset
- Also applies for the gradient

One full run through the training set is called an epoch

- Usually training takes many epochs

Resulting algorithm called Minibatch Gradient Descent

- Or Stochastic Gradient Descent (SGD) if $S = 1$
- In practice often called SGD even if $S > 1$

Time for single iteration is now independent of dataset size

In DL S varies between 1 and a few hundred samples

- Most common are 64, 128, 256
- 2^n for efficiency (data parallelism)

Decreasing S also decreases

- Computation time per iteration
- Memory required on GPU (minibatch processed as whole)
- Accuracy of the gradient estimate

Decreasing S causes more noisy gradient estimates

- Gives Gradient Descent ability to escape local minima [2]

Important to sample minibatches randomly

- To break (possible) ordering in dataset

Standard approach in practice

- Shuffle training set once or before every epoch
- Process sequentially in minibatches

Many alternatives

- Adagrad, RMSProp, Adam, ...
- Advantage of not having to choose the learning rate

Overall SGD with Nesterov momentum is the best choice

- Setting $\beta = 0.9$ is usually fine

Gradient Descent
Alternatives

Path finding comparison on challenging f

- Different learning rates, so speed not comparable

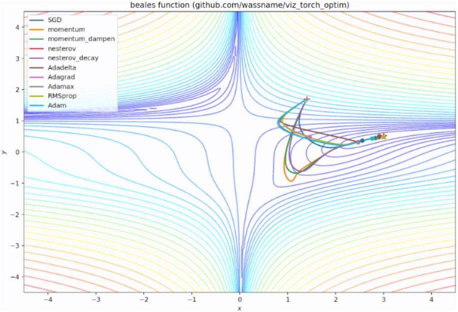


Image from github.com

Bibliography

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. 2016.

[2] Robert Kleinberg, Yuanzhi Li, and Yang Yuan. *An Alternative View: When Does SGD Escape Local Minima?* CoRR abs/1802.06175 (2018). URL: http://arxiv.org/abs/1802.06175.