Training Optimization I

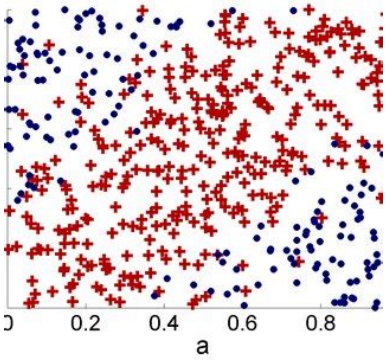
Based on "Deep Learning"

Penelope Mueck, Siba Mohsen

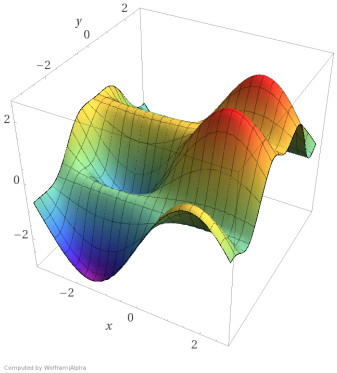
University of Bonn

08.12.2020

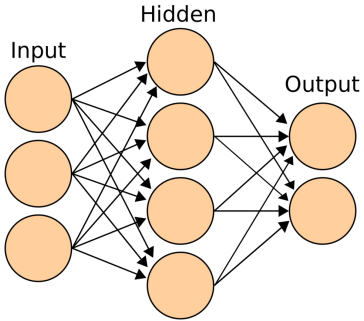
Motivation



[Lop16]



[Rez16]



[Dai16]

Outline

How Learning Differs from Pure Optimization

Training Deep Learning Models

- Optimize performance measure P defined w.r.t. test set
- P can only be optimized indirectly $\rightarrow$ minimize the **risk**

$$J^*(\theta) = E_{(x,y)\sim p_{data}}[L(f(x;\theta), y)]$$

  - p_{data}: data generating distribution
  - L: per-example loss function
  - $f(x;\theta)$: predicted output when input is x
  - y: target output
- p_{data} is unknown $\rightarrow$ minimize **empirical risk**

$$E_{(x,y)\sim \hat{p}_{data}}[L(f(x;\theta), y)] = \frac{1}{m}\sum_{i=1}^{m} L(f(x;\theta), y)$$

- **Empirical risk minimization** rarely used in deep learning
  - Loss functions do not have useful derivatives
  - Overfitting

Surrogate Loss Functions and Early Stopping

- Instead of the loss function we often minimize a **surrogate loss function**
- Minimizing the surrogate loss function halts when early stopping criterion is met
  - Training often halts when surrogate loss function still has large derivatives
- **Early stopping** criterion is based on true underlying loss function measured on the validation set

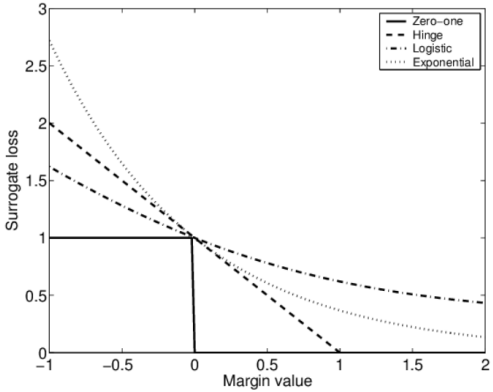


Figure: Surrogate loss functions for 0-1 loss [Ngu20].

Form of the Objective Function

- Objective function decomposes as a sum over training examples
- We compute each update to the parameters based on an expected value of the cost function
- Example: Maximum likelihood estimation

$$J(\theta) = E_{(x,y)\sim\hat{p}_{data}} \log p_{model}(x, y; \theta)$$
$$\nabla_\theta J(\theta) = E_{(x,y)\sim\hat{p}_{data}} \nabla_\theta \log p_{model}(x, y; \theta)$$

Batch, online and Stochastic Methods

- **Batch methods**: Optimization algorithms that use the entire training set
- **Online methods**: Optimization algorithms that use only a single example at a time
- **Minibatch/Stochastic methods**: Batch size between size for batch and online methods $\rightarrow$ used in deep learning

Stochastic Methods - How to Pick the Minibatches

- How to pick the minibatches:
  - Minibatches have to be selected randomly
  - Subsequent minibatches should be independent of each other
  - Shuffle examples if ordering is significant
  - Special case very large datasets: minibatches are constructed from shuffled examples rather than selected randomly
- Factors influencing the size:
  - Accuracy of estimate
  - Regularization vs. optimization
  - Hardware and memory
  - Multicore architectures are underutilized by very small batches $\rightarrow$ define minimum batch size

Stochastic Gradient Descent Minimizes Generalization Error

Assumptions:

- Examples are drawn from stream of data
- x and y are discrete $\Rightarrow$

$$J^*(\theta) = \sum_x \sum_y p_{data}(x, y) L(f(x; \theta), y)$$

$$\nabla_\theta J^*(\theta) = \sum_x \sum_y p_{data}(x, y) \nabla_\theta L(f(x; \theta))$$

$\Rightarrow \hat{g} = \frac{1}{m} \nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$ is an unbiased estimate of $\nabla_\theta J^*(\theta)$ if we sample a minibatch of examples $\{x^{(1)}, .., x^{(m)}\}$ with corresponding targets $y^{(i)}$ sampled from p_{data}

Challenges in Neural Optimization

Challenges Facing Optimization of Deep Neural Networks

- Ill-Conditioning
- Local Minima
- Plateaus, Saddle points and other Flat regions
- Cliffs
- Long-Term Dependencies
- Poor Correspondence between Local and Global Structure

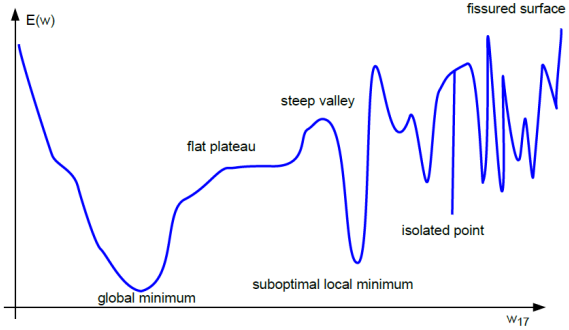


Figure: Loss function during training a neural network [Goe19].

Definitions (Recap)

- Given vector-valued function $f : \mathbb{R}^n \to \mathbb{R}^m$
- f consists of m functions $f_1, \ldots, f_m : \mathbb{R}^n \to \mathbb{R}$
- **Jacobian Matrix:** $J \in \mathbb{R}^{m \times n}$, $(J)_{i,j} = \frac{\partial f_i}{\partial x_j}$

$$J = \begin{bmatrix} | & & | \\ \nabla_x f_1 & \ldots & \nabla_x f_m \\ | & & | \end{bmatrix}$$

  $\to$ 1st-Order Optimization
- **Hessian Matrix:** $H \in \mathbb{R}^{n \times n}$, $H(f)(x)_{i,j} := \frac{\partial}{\partial x_i \partial x_j} f(x)$

  $\to$ 2nd-Order Optimization

Conditioning

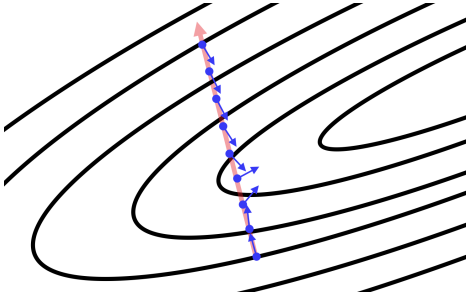


Figure: Gradient descent directions during training [source].

1. Neural Networks are trained by changing parameters based on an optimization algorithm (e.g. Stochastic Gradient Descent)
2. Optimization algorithm searches for local/global minima on loss function
3. Hessian matrix hints at curvature of functions (convex)
4. Condition number of the Hessian measures the difference between derivatives in each direction

Ill-Conditioning

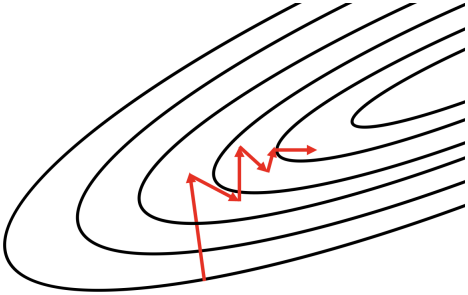


Figure: Gradient descent directions during training with ill-conditioned Hessian [source].

- **Challenges:** Poor conditioning imerges when the condition number is high:
  - gradient descent will perform poorly: which direction will the gradient choose?
  - choice of suitable step size becomes difficult: smaller steps to adapt to strong curvature $\rightarrow$ slow learning
- **Mitigation Techniques:**
  - Modification of Newton's method then applying it to the Neural Network
  - Momentum Algorithm

Local Minima

- Neural networks have nonconvex cost functions $\rightarrow$ several local minima
- Neural Networks are **nonidentifiable**, because there are many possibilities to select suitable weights during training
  - Infinite number of local minima
  - **Equivalent** to each other in cost value
  - Not problematic
- **Challenge:** Local minima have higher cost function value than global minimum
- **Mitigation Techniques:**
  - Most local minima present low cost function value
  - It is sufficient to find a convenient local minimum instead of the global minimum

Plateaus, Saddle Points and other Flat Regions

Saddle points:

- Most frequent in high dimensional nonconvex functions
- Can be local minimum and maximum of cost function depending on point of view
- How do 1^{st} and 2^{nd} order optimizations deal with saddle points?
  - 1^{st} order: The gradient becomes very small or escapes the point
  - 2^{nd} order: **Challenges:**
    1. The gradient may go directly and sit on the saddle point ($\nabla_x f(x) = 0$)
    2. Hard to be used in huge NN

  Mitigation Technique: Saddle-free Newton method by rapidly escaping high dimensional saddle points [Dau+14]

Plateaus and Flat Regions:

- Cause problems when optimizing nonconvex functions with no remediation techniques

Cliffs

- Dangerous from both sides: above and below
- **Challenge:** The gradient surpasses the cliff structure because it only determines which direction to choose and disregards step size
- **Mitigation Technique:** Gradient Clipping Heuristic (chapter 10) by reducing the step size to prohibit the gradient to surpass the cliff

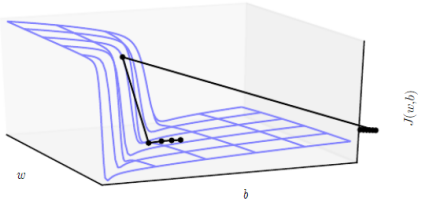


Figure: Cliff region [GBC16].

Long-Term Dependencies

- Very deep computational graphs caused by big number of layers in NN (e.g recurrent networks)
- **Challenges:** Vanishing and Exploding gradient descent
  - **Vanishing GD:** gradients don't know which direction to choose to improve the cost function
  - **Exploding GD:** makes the learning process inconsistent
- **Mitigation Technique:** Power method for recurrent and feedforward neural networks to discard uninteresting features in input vector

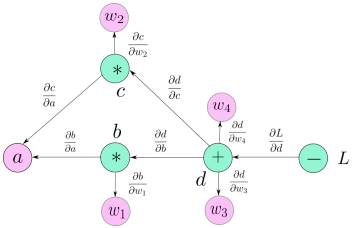


Figure: Computational Graph [source]

Poor Correspondence between Local and Global Structure

- Previous mitigation techniques solve the optimization problem at a single point on the loss function to arrive to a low cost value
- **Challenge:** Is this cost value sufficiently low w.r.t. other low values? Does this low value drives the point into a much lower cost value (e.g. global minimum)?
- **Mitigation Techniques:**
  - Force the gradient to start at good points on the loss function to get faster into a convenient minimum
  - Do not concentrate on finding the exact minimum of the loss function, rather try to achieve a low cost value that would generalize well

Basic Algorithms

SGD-Algorithm

Algorithm 1: Stochastic Gradient Descent (SGD) **update**

Require: Learning rate schedule ϵ_1,ϵ_2, ...

Require: Initial Parameter θ

Set $k = 0$;

while *stopping criterion is not met* **do**

  Pick a minibatch of m examples from the training set $\{\left(x^{(1)}, y^{(1)}\right), \ldots, \left(x^{(m)}, y^{(m)}\right)\}$;

  Compute gradient estimate: $\hat{g} = \frac{1}{m} \nabla_\theta \sum_i L\left(f\left(x^{(i)}; \theta\right), y^{(i)}\right)$;

  Apply update $\theta = \theta - \epsilon_k \hat{g}$;

  $k = k + 1$;

end

SGD-Learning Rate ϵ_k

- Tells how much to change the model based on the loss function
- Decreases over time
- To choose by trial and error or by depicting the learning curve over time
- In practice: for $\alpha = \frac{k}{\tau}$, decrease ϵ_k linearly until iteration τ:

$$\epsilon_k = (1 - \alpha)\epsilon_0 + \alpha\epsilon_\tau$$

  - $\tau =$ number of iterations to make few hundred passes through NN
  - $\epsilon_\tau = \frac{\epsilon_0}{100}$
  - $\epsilon_0 >$ best performing ϵ_k in the first iterations

SGD-Convergence and Computation

- Allows convergence even with huge number of training examples
- To calculate excess error for convergence: $J(\theta) - \min_\theta J(\theta)$
- SGD applied to a convex problem: excess error $= \mathcal{O}(\frac{1}{\sqrt{k}})$ after k iterations
- SGD applied to a strongly convex problem: excess error $= \mathcal{O}(\frac{1}{k})$ after k iterations

Momentum-Characteristics

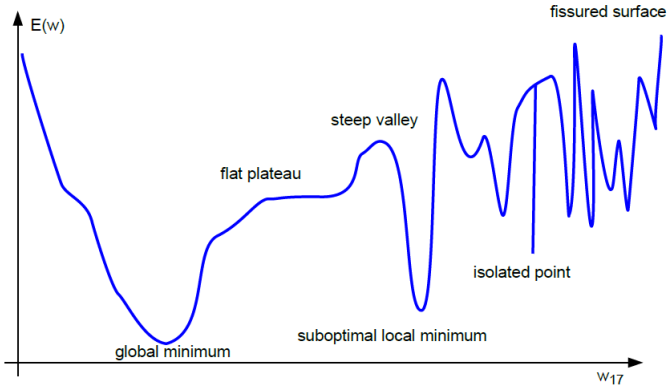


Figure: Loss function during training of a neural network [Goe19].

- Momentum in physics: mass $\times$ velocity
- Momentum is faster than SGD
- Momentum fixes variance problem in SGD caused by computing inexact derivates of the loss function
- Momentum is robust to high curvature and small/noisy gradients

Momentum-Algorithm

Algorithm 2: Stochastic Gradient Descent (SGD) **with momentum**

Require: Learning rate ϵ, momentum parameter α

Require: Initial Parameter θ, initial velocity v

while *stopping criterion is not met* **do**

Pick a minibatch of m examples from the training set $\{\left(x^{(1)}, y^{(1)}\right), \ldots, \left(x^{(m)}, y^{(m)}\right)\}$;

Compute gradient estimate: $g = \frac{1}{m} \nabla_\theta \sum_i L\left(f(x^{(i)}; \theta), y^{(i)}\right)$;

Compute velocity update: $v = \alpha v - \epsilon g$;

Apply update: $\theta = \theta + v$;

end

Momentum-Parameters

- Momentum algorithm accumulates a quickly decreasing average of past gradients and uses them in the next move
- Velocity v (momentum): direction and speed of parameters
- Momentum parameter $\alpha \in [0, 1)$: determines how quickly the contributions of previous gradients exponentially decrease and affect current move
- In practice: $\alpha \in 0.5, 0.9, 0.99$, increases over time
- $\theta(t)$: Point on the loss function at time t

Nesterov Momentum

- Adds *correction factor* to Momentum
- Gradient step is evaluated after application of momentum (velocity step)
- New update rule:

$$g = \frac{1}{m} \times \nabla_\theta \times \sum_i L\left(f(x^{(i)}; \theta + \alpha \mathrm{v}), y^{(i)}\right)$$

$$v = \alpha v - \epsilon g$$

$$\theta = \theta + v$$

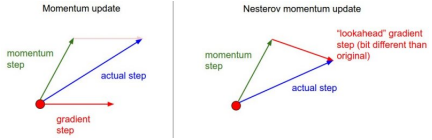


Figure: Momentum vs. Nesterov Momentum update step [source].

Parameter Initialization Strategies

Initialization for Deep Learning

- Training algorithms for deep learning are usually iterative $\rightarrow$ user has to specify an initial point
- Initial point affects
  - convergence
  - speed of convergence
  - if we converge to a point with high or low cost $\rightarrow$ points of comparable cost can have a different generalization error!

Characteristics of Initial Parameters

- Most initialization strategies are based on achieving good properties when the network is initialized
  - No good understanding of how these properties are preserved during training
  - Optimization vs. regularization
- Certainly known: Initial parameters need to **break symmetry** between different units
  - Hidden units with same activation function and connection to same input parameters must have different initial parameters

$\rightarrow$ Use **random initialization**

Random Initialization

- Weights are initialized randomly
- Values are drawn randomly from a Gaussian or uniform distribution
- Scale of initial distribution has a large effect on the outcome $\rightarrow$ influences optimization and generalization
  - Larger weights lead to stronger symmetry-breaking effect
  - Too large weights can cause exploding values during forward or backward-propagation or saturation of the activation function
  - Optimization perspective: weights should be large enough to propagate information successfully
  - Regularization: Keep weights small

Heuristics for Choosing Initial Scale of the Weights

1. Initialize weights by sampling each weight from $U\left(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}}\right)$
   - We assume we have a fully connected layer with m inputs and n outputs
2. Use **normalized initialization**: $W_{i,j} \sim U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right)$
3. Initialize to random orthogonal matrices with **gain** factor g that needs to be carefully chosen
4. Use **sparse initialization**: each unit is initialized to have exactly k nonzero weights

- Optimal criteria for initial weights do not lead to optimal performance
  - Treat initial weights as hyperparameters
  - Treat initial scale of the weights and whether to use sparse or dense initialization as hyperparameter if not too costly

Initializing the Biases

- Approach for setting the biases must be coordinated with the approach for setting the weights
- Setting the biases to zero is compatible with most weight initialization schemes
- Cases where biases may be set to nonzero values:
  - If a bias is for an output unit $\rightarrow$ beneficial to initialize the bias to obtain the right marginal statistics of the output
  - When we want to avoid too much saturation at initialization
  - When a unit controls whether other units are able to participate in a function

Questions

White Board

White Board

Example: Long-Term Dependencies

- Suppose that a path of the computational graph applies a repeated multiplication with a matrix $\mathbf{W}$, where $\mathbf{W} = \mathbf{V}diag(\lambda)\mathbf{V}^{-1}$ is the eigendecomposition of $\mathbf{W}$.
- After t multiplication steps, we are multiplying by $\mathbf{W}^t$ and the eigendecomposition becomes $\mathbf{W}^t = \mathbf{V}diag(\lambda)^t\mathbf{V}^{-1}$
- The Vanishing and Exploding gradient descent problem arises from scaling $diag(\lambda)$.
- The Power Method can be deployed to detect the largest eigenvalue λ_i of $\mathbf{W}$ and its eigenvector and then to rule out all components that are orthogonal to $\mathbf{W}$.

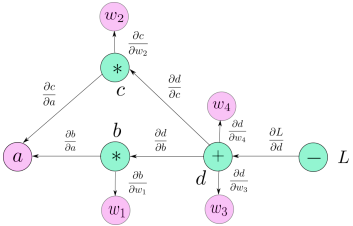


Figure: Computational Graph [source]

References

Yann N Dauphin et al. "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization". In: *Advances in neural information processing systems*. 2014, pp. 2933–2941.

Shaumik Daityari. "A Beginners Guide to Keras". In: (2016). URL: %5Curl%7Bhttps://www.sitepoint.com/keras-digit-recognition-tutorial/%7D.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

Emilia Lopez-Inesta. In: (2016). URL: %5Curl%7Bhttps://www.researchgate.net/profile/Emilia_Lopez-Inesta%7D.

Reza. "The Hard Thing in Deep Learning". In: (2016). URL: %5Curl%7Bhttps://www.matroid.com/blog/post/the-hard-thing-about-deep-learning%7D.

Dr. Nils Goerke. "TNN$_W$S19$_0$3$_T$raining$_M$LPs$_{wB}$P$_s$lides.pdf". In: (2019). URL: https://www.ais.uni-bonn.de/WS1920/4204_L_NN.html.