Deep Learning

4. Optimization for Training Deep Models

Lars Schmidt-Thieme

Information Systems and Machine Learning Lab (ISMLL)
Institute for Computer Science
University of Hildesheim, Germany

Syllabus

Tue. 21.4.	(1)	1. Supervised Learning (Review 1)
Tue. 28.4.	(2)	2. Neural Networks (Review 2)
Tue. 5.5.	(3)	3. Regularization for Deep Learning
Tue. 12.5.	(4)	4. Optimization for Training Deep Models
Tue. 19.5.	(5)	5. Convolutional Neural Networks
Tue. 26.5.	(6)	6. Recurrent Neural Networks
Tue. 2.6.	—	— *Pentecoste Break* —
Tue. 9.6.	(7)	7. Autoencoders
Tue. 16.6.	(8)	8. Generative Adversarial Networks
Tue. 23.6.	(9)	9. Recent Advances
Tue. 30.6.	(10)	10. Engineering Deep Learning Models
Tue. 7.7.	(11)	tbd.
Tue. 14.7.	(12)	Q & A

Outline

Outline

1. Learning as Optimization

2. Parameter Initializations

3. Gradient Estimation and Momentum

4. Adaptive Learning Rates

Learning as Optimization

- Optimization:
  find the parameters x^* with minimum value of the objective function f:

$$x^* := \arg\min_x f(x)$$

- Learning:
  find the model parameters θ^* with minimum value of the objective function f for the training data set:

$$\theta^* := \arg\min_\theta f(\theta; \mathcal{D}^{\text{train}})$$

$$f(\theta; \mathcal{D}^{\text{train}}) := \frac{1}{N}\left(\sum_{n=1}^{N} \ell(y_n, \hat{y}(x_n; \theta))\right) + \lambda\Omega(\theta),$$

$$= \frac{1}{N}\sum_{n=1}^{N} f(\theta; \{(x_n, y_n)\})$$

$$\mathcal{D}^{\text{train}} = \{(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\}$$

Gradient Descent (basic version)

learn-gd$(f : \mathbb{R}^P \rightarrow \mathbb{R}, \mathcal{D}^{\text{train}}, \sigma^2 \in \mathbb{R}^+, \mu, i_{\max} \in \mathbb{N})$:
  $\theta \sim \mathcal{N}(0, \sigma^2)$
  for $i = 1, \ldots, i_{\max}$:
    $g := \nabla f(\theta; \mathcal{D}^{\text{train}})$
    $\theta := \theta - \mu_i g$
  return θ

f	objective function (as function in the parameters θ)
$\mathcal{D}^{\text{train}}$	training data
σ^2	parameter initialization variance
μ	step size schedule
$i_{\max}$	maximal number of iterations

Issues: Non-Convexity / Local Minima

- The objective functions of neural networks are highly non-convex

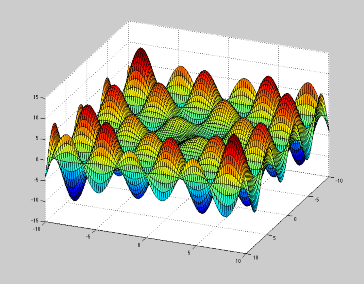


Figure 1: A non-convex function has multiple local minima, source: `imgur.com`

Issues: Saddle Points

- In addition to local minima, objective functions include saddle points

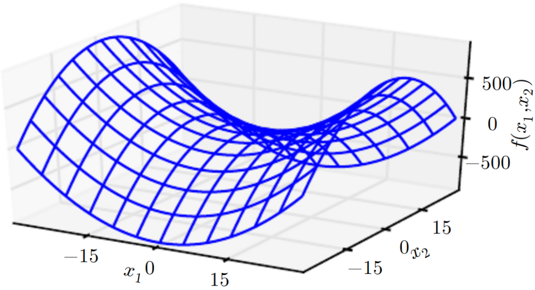
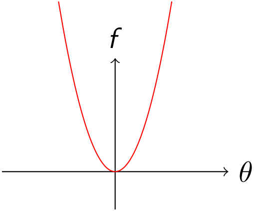


Figure 2: Saddle points, Source: Goodfellow et al., 2016

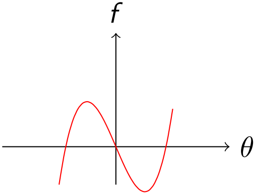
- Gradients are very small around a saddle point

Convex vs. Non-Convex Objective Functions

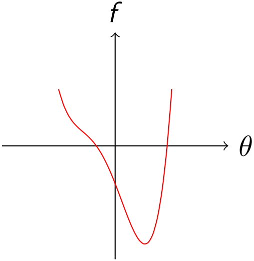
A. $f(\theta) := \theta^2$



B. $f(\theta) := \frac{1}{3}\theta^3 - \frac{7}{3}\theta$



C.
$f(\theta) := \frac{37}{360}\theta^4 + \frac{17}{60}\theta^3 - \frac{133}{360}\theta^2 - \frac{51}{20}\theta - 2$



Outline

Parameter Initializations May Matter

- convex optimization:
  - there exists a global minimum
  - it does not matter where we start,
    minimization always converges to the global minimum
    - e.g., initialize $\theta = 0$
- non-convex optimization:
  - there exist many local minima
  - depending on where we start,
    minimization might converge to a different local minimum

    $\rightsquigarrow$ parameter initialization may matter

Issue: Symmetric Networks Stay Symmetric

- neural network:

$$z^\ell(z^{\ell-1}) := a(W^\ell z^{\ell-1} + b^\ell)$$

- assume we initialized all neurons of each layer with the same weights and biases:

$$W^\ell_{m,k} = W^\ell_{m',k}, \quad b^\ell_m = b^\ell_{m'} \quad \forall \ell, m, m', k$$

- then their gradients are identical:

$$\frac{\partial f(\theta)}{\partial W^\ell_{m,k}} = \frac{\partial f(z^{L+1})}{\partial z^\ell} \frac{\partial z^\ell(z^{\ell-1})}{\partial W^\ell_{m,k}}$$

$\rightsquigarrow$ we need to **break the symmetry**
- e.g., initialize randomly $W^\ell_{m,k} \sim \mathcal{N}(0, \sigma^2)$

Normalized Initialization [Glorot and Bengio, 2010]

- keep variances of all layers and all gradients constant:
  - view all variables X, Z^ℓ, W^ℓ as random variables
  - assume no activation function, independent weight matrices W^ℓ

$$\mathrm{var}(Z^\ell) = \mathrm{var}(W^\ell)\, M_{\ell-1}\, \mathrm{var}(Z^{\ell-1}) \qquad \rightsquigarrow \mathrm{var}(W^\ell) \stackrel{!}{=} \frac{1}{M_{\ell-1}}$$

$$\mathrm{var}(\nabla_{Z^{\ell-1}} f) = \mathrm{var}(W^\ell)\, M_\ell\, \mathrm{var}(\nabla_{Z^\ell} f) \qquad \rightsquigarrow \mathrm{var}(W^\ell) \stackrel{!}{=} \frac{1}{M_\ell}$$

- Q: How should we set the variance of W such that both,
  - the variance of the latent values z and
  - the variance of the gradients

  stays constant across layers?

Normalized Initialization [Glorot and Bengio, 2010]

- keep variances of all layers and all gradients constant:
  - view all variables X, Z^ℓ, W^ℓ as random variables
  - assume no activation function, independent weight matrices W^ℓ

$$\mathrm{var}(Z^\ell) = \mathrm{var}(W^\ell)\, M_{\ell-1}\, \mathrm{var}(Z^{\ell-1}) \qquad \rightsquigarrow \mathrm{var}(W^\ell) \stackrel{!}{=} \frac{1}{M_{\ell-1}}$$

$$\mathrm{var}(\nabla_{Z^{\ell-1}} f) = \mathrm{var}(W^\ell)\, M_\ell\, \mathrm{var}(\nabla_{Z^\ell} f) \qquad \rightsquigarrow \mathrm{var}(W^\ell) \stackrel{!}{=} \frac{1}{M_\ell}$$

- weights:

$$W^\ell_{m,k} \sim \mathrm{unif}(-\sqrt{\frac{6}{M_{\ell-1}+M_\ell}}, \sqrt{\frac{6}{M_{\ell-1}+M_\ell}}), \quad \text{with layer sizes } M_\ell$$

  - uniform distribution has variance

$$\mathrm{var}(\mathrm{unif}(a,b)) = \frac{(b-a)^2}{12} \underset{\text{here}}{=} \frac{2}{M_{\ell-1}+M_\ell} = \frac{1}{\frac{M_{\ell-1}+M_\ell}{2}} \text{ a compromise}$$

- biases: $b^\ell_m := 0$

Initializing Biases

- biases often just set to zero
- biases on hidden layers:
  - set to a small positive constant:

$$b^\ell := c > 0$$

  - e.g., $c := 0.1$
  - esp. for ReLU to avoid the "Dead ReLU" phenomenon
- biases on output layer:
  - optimal biases to predict average output for zero weights:

$$b^{L+1} := \arg\min_b \ell(\bar{y}, 0 + b), \quad \bar{y} := \frac{1}{N}\sum_{n=1}^{N} y_n$$

Outline

Gradient Estimation

- global shape of the objective function is unknown
- only have local gradients as information:
  - **batch** of full training set $\mathcal{D}^{\text{train}} := \{(x_1, y_1), \ldots, (x_N, y_N)\}$:

$$g := \nabla_\theta f(\theta; \mathcal{D}^{\text{train}}) = \frac{1}{N} \sum_{n=1}^{N} \nabla_\theta f(\theta; x_n, y_n)$$

  - **mini batch** $\mathcal{D}^{\text{batch}} = \{(x_{n_1}, y_{n_1}), \ldots, (x_{n_B}, y_{n_B})\} \subseteq \mathcal{D}^{\text{train}}$ for $B \ll N$:

$$\hat{g} := \nabla_\theta f(\theta; \mathcal{D}^{\text{batch}}) = \frac{1}{|\mathcal{D}^{\text{batch}}|} \sum_{(x,y) \in \mathcal{D}^{\text{batch}}} \nabla_\theta f(\theta; x, y)$$

$$= \frac{1}{B} \sum_{b=1}^{B} \nabla_\theta f(\theta; x_{n_b}, y_{n_b})$$

  - **online** w.r.t. a single instance (x_n, y_n) (= mini batch with $B = 1$):

$$\hat{g} = \nabla_\theta f(\theta; x_n, y_n)$$

Stochastic Gradient Descent (basic version)

```
learn-sgd(f : ℝ^P → ℝ, D^train, σ² ∈ ℝ⁺, μ, i_max ∈ ℕ, B ∈ ℕ):
  θ ∼ N(0, σ²)
  for i = 1, ..., i_max:
    D^batch ∼ D^train draw B instances uniformly at random
    g := ∇f(θ; D^batch)
    θ := θ − μ_i g
  return θ
```

f objective function (as function in the parameters θ)

$\mathcal{D}^{\text{train}}$ training data

σ^2 parameter initialization variance

μ step size schedule

$i_{\max}$ maximal number of iterations

B minibatch size

Momentum

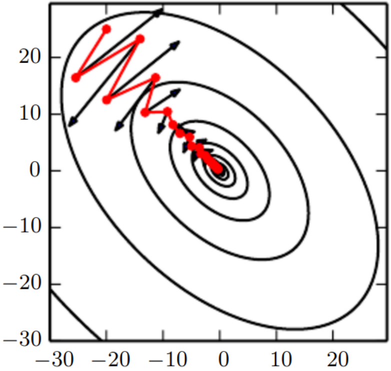


Figure 3: A quadratic loss with a poor conditioned Hessian; Black arrows: Gradient descent steps; Red line: Momentum correction, Source: Goodfellow et al., 2016

Momentum

- instead of only using the current gradient in each step

  $$g_1 := \nabla_\theta f(\theta_1; \mathcal{D}_1^{\text{batch}}), g_2 := \nabla_\theta f(\theta_2; \mathcal{D}_2^{\text{batch}}), \ldots, g_i := \nabla_\theta f(\theta_i; \mathcal{D}_i^{\text{batch}})$$

  use an **exponentially smoothed update step**:

  $$\begin{aligned} v &:= -\sum_{j=1}^{i} \alpha^{i-j} \mu_j g_j \\ &= -\alpha^{i-1}\mu_1 g_1 - \alpha^{i-2}\mu_2 g_2 - \cdots - \alpha^1 \mu_{i-1} g_{i-1} - \cancel{\alpha^0} \mu_i g_i, \quad \alpha \in [0,1) \end{aligned}$$

- Q: How can we compute v efficiently?

Momentum

- instead of only using the current gradient in each step

  $$g_1 := \nabla_\theta f(\theta_1; \mathcal{D}_1^{\text{batch}}), g_2 := \nabla_\theta f(\theta_2; \mathcal{D}_2^{\text{batch}}), \ldots, g_i := \nabla_\theta f(\theta_i; \mathcal{D}_i^{\text{batch}})$$

  use an **exponentially smoothed update step**:

  $$\begin{aligned} v &:= -\sum_{j=1}^{i} \alpha^{i-j} \mu_j g_j \\ &= -\alpha^{i-1}\mu_1 g_1 - \alpha^{i-2}\mu_2 g_2 - \cdots - \alpha^1 \mu_{i-1} g_{i-1} - \cancel{\alpha^0} \mu_i g_i, \quad \alpha \in [0,1) \end{aligned}$$

- v can be computed efficiently recursively:

  $$v := \alpha v - \mu_i g_i$$

- finally $\theta := \theta + v$

- αv is often called **momentum**, v sometimes a velocity.

SGD with Momentum

learn-sgd-moment$(f : \mathbb{R}^P \rightarrow \mathbb{R}, \mathcal{D}^{\text{train}}, \sigma^2 \in \mathbb{R}^+, \mu, i_{\max} \in \mathbb{N}, B \in \mathbb{N}, \alpha)$:

  $\theta \sim \mathcal{N}(0, \sigma^2)$

  $v := 0$

  for $i = 1, \ldots, i_{\max}$:

    $\mathcal{D}^{\text{batch}} \sim \mathcal{D}^{\text{train}}$ draw B instances uniformly at random

    $g := \nabla f(\theta; \mathcal{D}^{\text{batch}})$

    $v := \alpha v - \mu_i g$

    $\theta := \theta + v$

  return θ

α update step decay factor, e.g., $\alpha \in \{0.5, 0.9, 0.99\}$

Nesterov Momentum

- Nesterov momentum adds a correction (look-ahead) factor to the standard velocity update

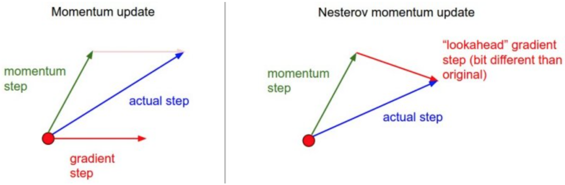


Figure 4: Nesterov momentum with a correction factor

- Computes gradient at the updated weights:

$$v := \alpha v - \mu \nabla_\theta f(\theta + \alpha v; \mathcal{D}^{\text{batch}})$$

- currently the most widely used momentum in Deep Learning libraries (Tensorflow, PyTorch)

SGD with Nesterov Momentum

learn-sgd-nesterov$(f : \mathbb{R}^P \rightarrow \mathbb{R}, \mathcal{D}^{\text{train}}, \sigma^2 \in \mathbb{R}^+, \mu, i_{\text{max}} \in \mathbb{N}, B \in \mathbb{N}, \alpha)$:

  $\theta \sim \mathcal{N}(0, \sigma^2)$

  $v := 0$

  for $i = 1, \ldots, i_{\text{max}}$:

    $\mathcal{D}^{\text{batch}} \sim \mathcal{D}^{\text{train}}$ draw B instances uniformly at random

    $g := \nabla f(\theta + \alpha v; \mathcal{D}^{\text{batch}})$

    $v := \alpha v - \mu_i g$

    $\theta := \theta + v$

  return θ

α update step decay factor, e.g., $\alpha \in \{0.5, 0.9, 0.99\}$

Outline

Stochastic Gradient Descent and Learning Rates

- What is a good learning rate / step size μ ?

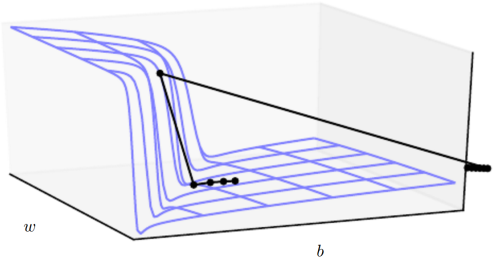


Figure 5: Cliffs and Exploding Gradients, Source: Goodfellow et al., 2016

Decaying Learning Rates

- Converges if $\sum_{i=1}^{\infty} \mu_i = \infty$ and $\sum_{i=1}^{\infty} \mu_i^2 < \infty$
- In practice, it is common to decay the learning rate:

$$\mu_i = \begin{cases} \left(1 - \frac{i}{\tau}\right) \mu_0 + \frac{i}{\tau} \mu_\tau & \text{if } i < \tau \\ \mu_\tau & \text{if } i \geq \tau \end{cases}, \quad \text{where } \mu_0 \gg \mu_\tau$$

Adagrad

- individual learning rate $\tilde{\mu}_{i,p}$ for every iteration i and parameter θ_p
- strongly decrease learning rate for large gradients:

$$r_p := r_p + g_p^2$$
$$\tilde{\mu}_{i,p} := \frac{\mu_i}{\delta + \sqrt{r_p}}, \quad p = 1, \ldots, P$$

  - $\delta > 0$ a small constant
- rapid progress in gently sloped directions
- Q: What might happen if Adagrad is run for many iterations?

Note: In vector notation: $r := r + g \odot g$ and $\tilde{\mu}_i := \frac{\mu_i}{\delta + \sqrt{r}}$ where $\odot$ is elementwise product and $\sqrt{r}$ also taken elementwise.

SGD with Adagrad

learn-sgd-adagrad$(f : \mathbb{R}^P \rightarrow \mathbb{R}, \mathcal{D}^{\text{train}}, \sigma^2 \in \mathbb{R}^+, \mu, i_{\max} \in \mathbb{N}, B \in \mathbb{N}, \delta)$:

  $\theta \sim \mathcal{N}(0, \sigma^2)$

  $r := 0$

  for $i = 1, \ldots, i_{\max}$:

    $\mathcal{D}^{\text{batch}} \sim \mathcal{D}^{\text{train}}$ draw B instances uniformly at random

    $g := \nabla f(\theta; \mathcal{D}^{\text{batch}})$

    $r := r + g \odot g$

    $\tilde{\mu} := \frac{\mu_i}{\delta + \sqrt{r}}$

    $\theta := \theta - \tilde{\mu} \odot g$

  return θ

$\delta > 0$ small constant

Root Mean Square Propagation (RMSProp)

- As $\sqrt{r}$ monotonically increases in Adagrad, $\frac{\mu}{\sqrt{r}}$ becomes too small
- RMSProp introduces an exponentially decaying average of the squared gradient history

learn-sgd-rmsprop$(f : \mathbb{R}^P \rightarrow \mathbb{R}, \mathcal{D}^{\text{train}}, \sigma^2 \in \mathbb{R}^+, \mu, i_{\max} \in \mathbb{N}, B \in \mathbb{N}, \delta, \rho)$:

  $\theta \sim \mathcal{N}(0, \sigma^2)$

  $r := 0$

  for $i = 1, \ldots, i_{\max}$:

    $\mathcal{D}^{\text{batch}} \sim \mathcal{D}^{\text{train}}$ draw B instances uniformly at random

    $g := \nabla f(\theta; \mathcal{D}^{\text{batch}})$

    $r := \rho r + (1 - \rho) g \odot g$

    $\tilde{\mu} := \frac{\mu_i}{\delta + \sqrt{r}}$

    $\theta := \theta - \tilde{\mu} \odot g$

  return θ

SGD with Nesterov Momentum and RMSProp

```
learn-sgd-nesterov-rmsprop(f : ℝ^P → ℝ, D^train, σ² ∈ ℝ⁺, μ, i_max ∈ ℕ, B ∈ ℕ, α, ρ):
  θ ~ N(0, σ²)
  v := 0
  r := 0
  for i = 1, ..., i_max:
    D^batch ~ D^train draw B instances uniformly at random
    g := ∇f(θ+αv; D^batch)
    r := ρr + (1 − ρ)g ⊙ g
    μ̃ := μ_i / (δ + √r)
    v := αv − μ̃ ⊙ g
    θ := θ + v
  return θ
```

α update step decay factor, e.g., $\alpha \in \{0.5, 0.9, 0.99\}$

ρ gradient square decay factor

SGD with Adaptive Moment (ADAM)

learn-sgd-adam$(f : \mathbb{R}^P \rightarrow \mathbb{R}, \mathcal{D}^{\text{train}}, \sigma^2 \in \mathbb{R}^+, \mu, i_{\max} \in \mathbb{N}, B \in \mathbb{N}, \alpha, \rho)$:
  $\theta \sim \mathcal{N}(0, \sigma^2)$
  $v := 0$
  $r := 0$
  for $i = 1, \ldots, i_{\max}$:
    $\mathcal{D}^{\text{batch}} \sim \mathcal{D}^{\text{train}}$ draw B instances uniformly at random
    $g := \nabla f(\theta; \mathcal{D}^{\text{batch}})$
    $v := \frac{1}{1-\alpha^i}(\alpha v + (1-\alpha) g)$
    $r := \frac{1}{1-\rho^i}(\rho r + (1-\rho) g \odot g)$
    $\tilde{\mu} := \frac{\mu_i}{\delta + \sqrt{r}}$
    $\theta := \theta - \tilde{\mu} v$
  return θ

α gradient decay factor, e.g., $\alpha \in \{0.5, 0.9, 0.99\}$
ρ gradient square decay factor

Comparing Various Optimization Approaches

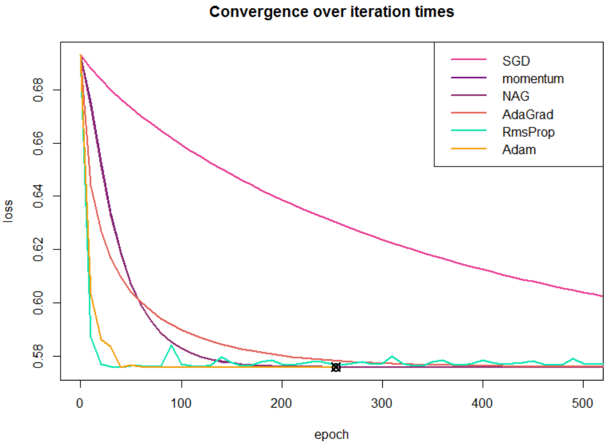


Figure 6: Optimizing a logistic regression model, Source: `gmo.jp`

Illustrations of Performance

Two illustrations (Source: cs.stanford.edu)

- `http://cs231n.github.io/assets/nn3/opt1.gif`
- `http://cs231n.github.io/assets/nn3/opt2.gif`

Summary (1/2)

- **Learning** the parameters of a model means **minimizing the objective function**.
  - the objective function is a big sum over instance wise losses and a regularization term
  - a stochastic function estimated based on **mini batches** (subsets of the training data)
  - gradients then also are averages over mini batches
- Learning a neural network is a highly **non-convex optimization problem**
  - many local minima
  - saddle points
- **Parameter initialization** matters.
  - it must be randomized (to **break the symmetry**)
  - **normalized initialization** to enforce similar variances of latent values and gradients across layers

Summary (2/2)

- a **momentum** can be added to SGD to stabilize the search direction
  - sum of exponetially decayed update steps
    (instead of just last update step)
  - **Nesterovs momentum**: look ahead
- **learning rates** can be computed adaptively
  - individual for each parameter (**AdaGrad**, **RMSProp**)
  - momentum and adaptive learning rates can be combined (**ADAM**)

Further Readings

- Goodfellow et al. 2016, ch. 8
- for initialization: Zhang et al. 2020, ch. 4.8
- lecture Modern Optimization Techniques, chapters 2.1 and 2.2.

Acknowledgement: An earlier version of the slides for this lecture have been written by my former postdoc Dr Josif Grabocka.

References

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 249–256, 2010.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The Mit Press, Cambridge, Massachusetts, November 2016. ISBN 978-0-262-03561-3.

Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander Smola. *Dive into Deep Learning*. https://d2l.ai/, 2020.