# Optimization for Deep Learning

# Agenda

1. Introduction
2. Gradient descent variants
3. Challenges
4. Gradient descent optimization algorithms
5. Parallelizing and distributing SGD
6. Additional strategies for optimizing SGD
7.

# Introduction

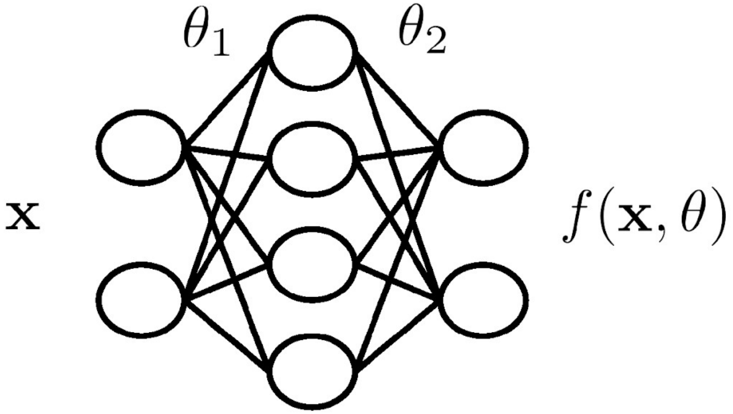
## Empirical Risk Minimization (ERM)

- Given training set $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$
- Prediction function $f(\mathbf{x}_i, \theta) \in \mathbb{R}$ parameterized by $\theta$
- **Empirical risk minimization:** Find a paramater that minimizes the loss function

$$\min_{\theta} \frac{1}{n} \sum_{i=1}^{n} \ell(f(\mathbf{x}_i, \theta), y_i) := L(\theta)$$

where $\ell(\cdot, \cdot)$ is a loss function e.g., MSE, cross entropy,

- For example, neural network has $f(\mathbf{x}, \theta) = \theta_k^\top \sigma\left(\theta_{k-1}^\top \sigma(\cdots \sigma(\theta_1^\top \mathbf{x}))\right)$



$$L(\theta) = \frac{1}{n} \sum_i (f(\mathbf{x}_i, \theta) - y_i)^2$$

Next, how to solve ERM?

# Introduction

- Gradient descent is a way to minimize an objective function $J(\theta)$
  - $\theta \in \mathbb{R}^d$: model parameters
  - $\eta$: learning rate
  - $\nabla_\theta J(\theta)$: gradient of the objective function with regard to the parameters
- Updates parameters **in opposite direction** of gradient.
- Update equation: $\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$

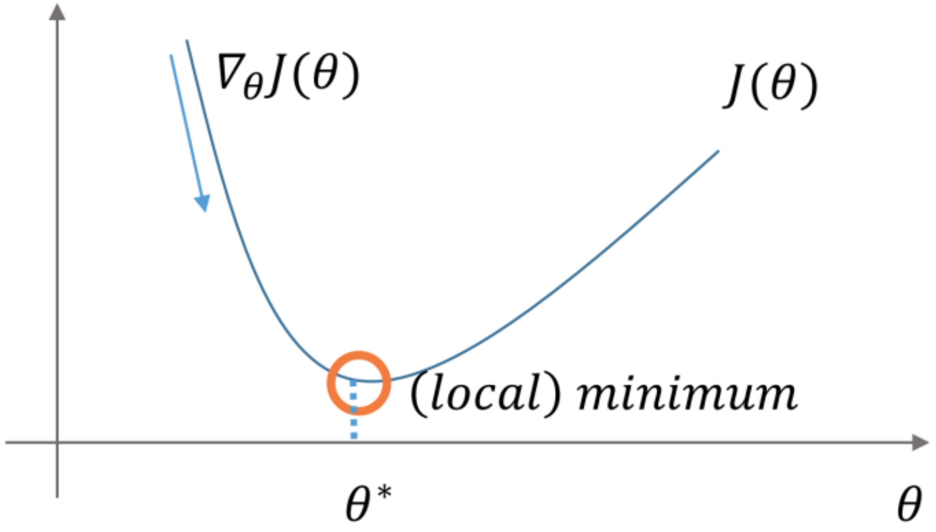


Figure: Optimization with gradient descent

# Gradient descent variants

1. Batch gradient descent
2. Stochastic gradient descent
3. Mini-batch gradient descent

Difference: Amount of data used per update

# Batch gradient descent

- Computes gradient with the **entire** dataset.
- Update equation: $\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$

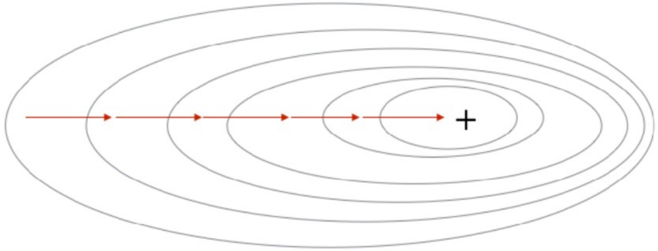
```python
for i in range(nb_epochs):
  params_grad = evaluate_gradient(
    loss_function, data, params)
  params = params - learning_rate * params_grad
```

Listing 1: Code for batch gradient descent update

# Batch gradient descent

- **Gradient descent (GD)** updates parameters iteratively by taking gradient.

$$\theta_{t+1} = \overset{\text{parameters}}{\theta_t} - \underset{\text{learning rate}}{\gamma}\underbrace{\nabla \overset{\text{loss function}}{L}\left(\theta_t\right)}_{:= \frac{1}{n}\sum_{i=1}^{n} \nabla \ell(\theta_t; \mathbf{x}_i, y_i)}$$



- (+) Converges to global (local) minimum for convex (non-convex) problem.
- (−) Not efficient with respect to computation time and memory space for huge $n$.
- For example, ImageNet dataset has $n$ =**1,281,167** images for training.

1.2M of 256x256 RGB images
≈ 236 GB memory

Next, efficient GD

- Pros:
  - Guaranteed to converge to **global** minimum for **convex** error surfaces and to a **local** minimum for **non-convex** surfaces.
- Cons:
  - **Very slow**.
  - Intractable for datasets that **do not fit in memory**.
  - **No online learning**.

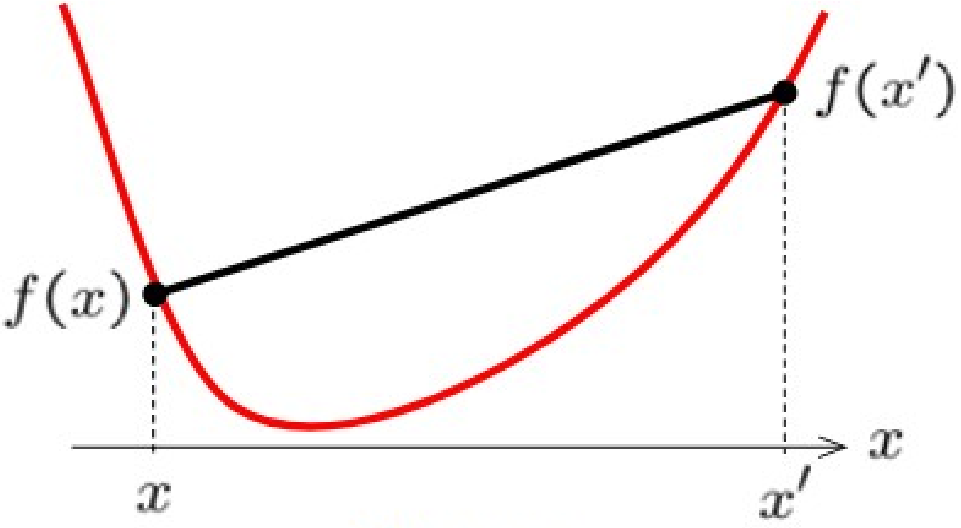
# Convex functions

A function $f : A \subseteq \mathbb{X} \to \mathbb{R}$ defined on a convex set $A$ is called **convex** if

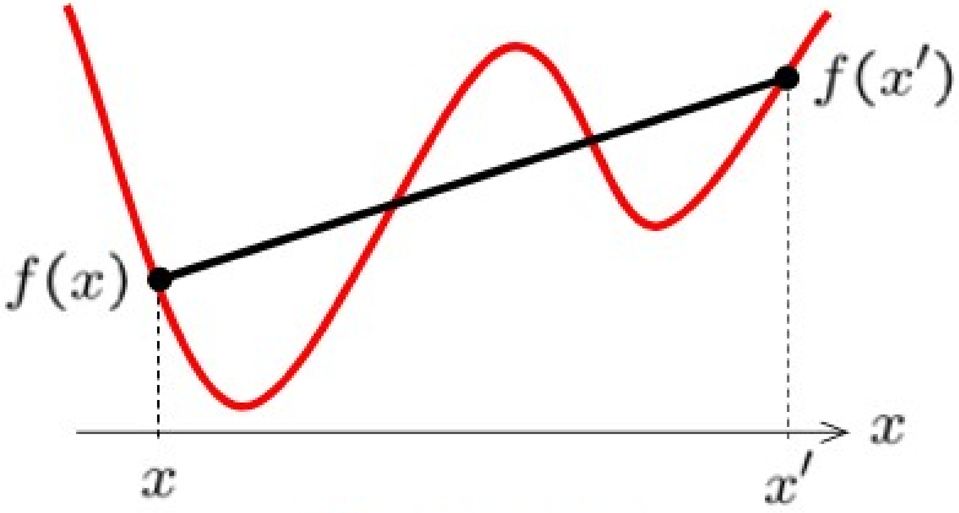
$$f(\lambda x + (1-\lambda)x') \leq \lambda f(x) + (1-\lambda) f(x')$$

for any $x, x' \in \mathbb{X}$ and $\lambda \in [0, 1]$

For convex function local minimum = global minimum



Convex



Non-convex

# Stochastic gradient descent

- Computes update for **each** example $x^{(i)}y^{(i)}$.
- Update equation: $\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for example in data:
    params_grad = evaluate_gradient(
      loss_function, example, params)
    params = params - learning_rate * params_grad
```

Listing 2: Code for stochastic gradient descent update

- Pros
  - **Much faster** than batch gradient descent.
  - Allows **online learning**.
- Cons
  - **High variance** updates.

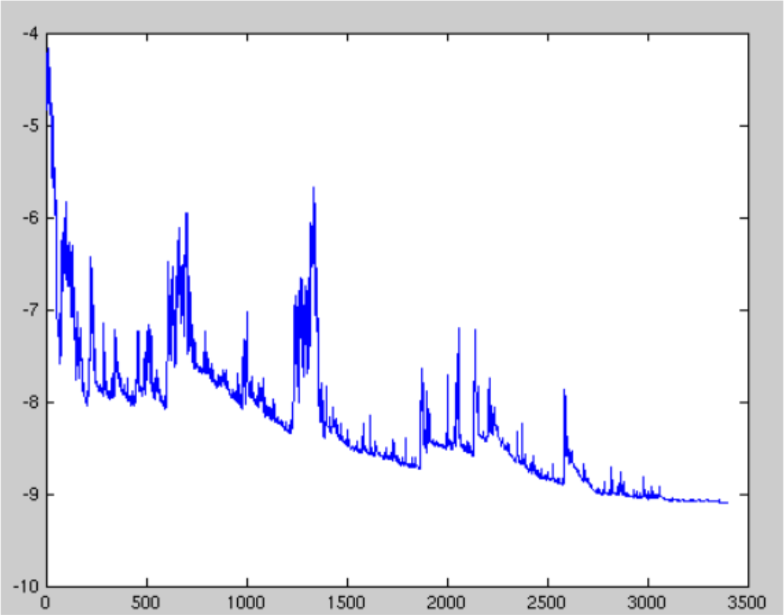


Figure: SGD fluctuation (Source: Wikipedia)

# Batch gradient descent vs. SGD fluctuation

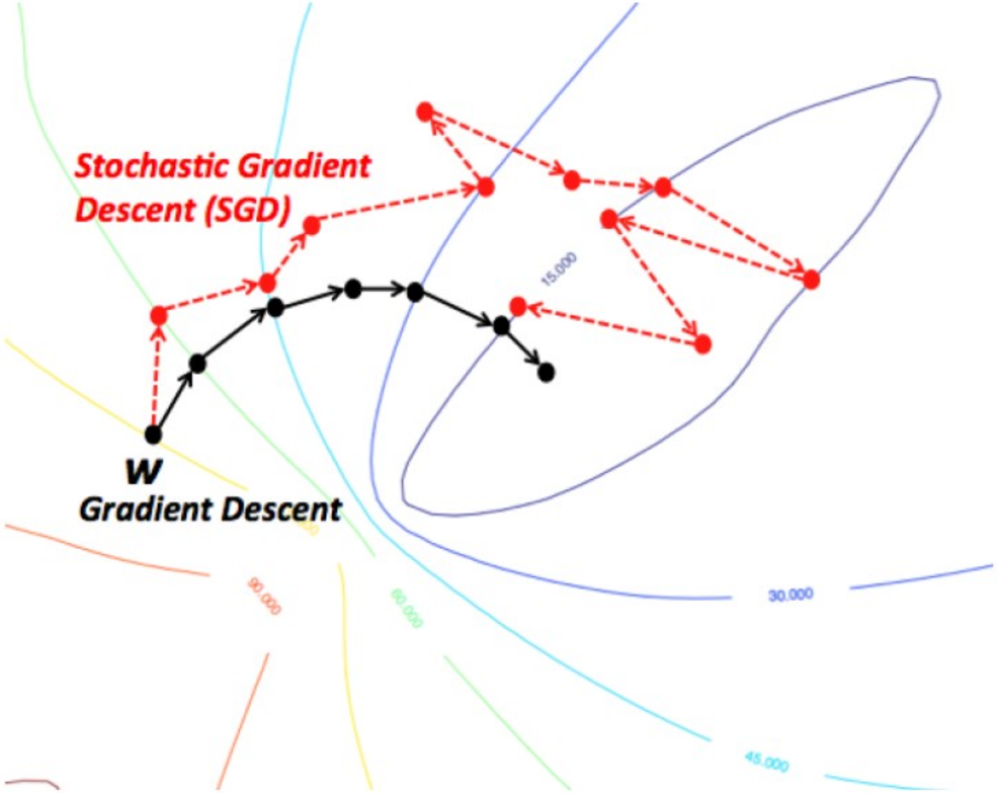


Figure: Batch gradient descent vs. SGD fluctuation (Source: wikidocs.net)

- SGD shows same convergence behaviour as batch gradient descent if learning rate is **slowly decreased (annealed)** over time.

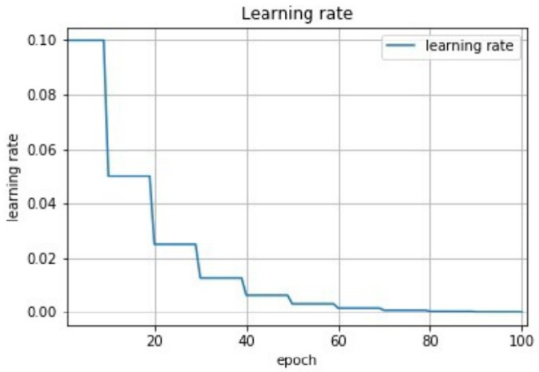
# Stochastic gradient descent

Learning rate scheduling : decay methods

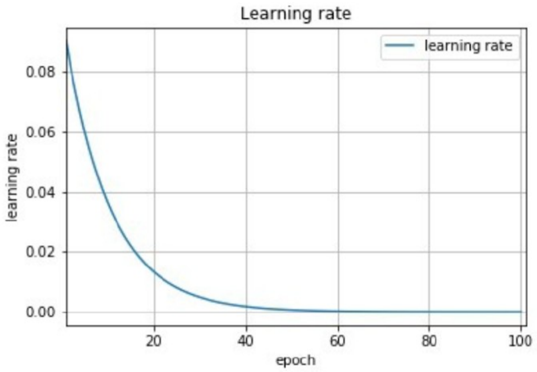
- A naive choice is the **constant** learning rate
- Common learning rate schedules include time-based/step/exponential decay

| | Time-based | Exponential | Step (most popular in practice) |
|---|---|---|---|
| $\gamma_t$ | $\dfrac{\gamma_0}{1+kt}$ | $\gamma_0 \exp(-kt)$ | $\gamma_0 \exp(-k\lfloor \dfrac{t}{T_{\text{epoch}}} \rfloor)$ |

- "Step decay" decreases learning rate by a factor every few epochs
- Typically, it is set $\gamma_0 = 0.01$ and drops by half ever $T_{\text{epoch}} = 10$ epoch



**step decay**



**exponential decay**

# Mini-batch gradient descent

- Performs update for every **mini-batch** of $n$ examples.
- Update equation: $\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(
      loss_function, batch, params)
    params = params - learning_rate * params_grad
```

Listing 3: Code for mini-batch gradient descent update

- Pros
  - **Reduces variance** of updates.
  - Can exploit **matrix multiplication** primitives.
- Cons
  - **Mini-batch size** is a hyperparameter. Common sizes are 50-256.
- Typically the algorithm of choice.
- Usually referred to as SGD even when mini-batches are used.

| Method | Accuracy | Update Speed | Memory Usage | Online Learning |
|---|---|---|---|---|
| **Batch** gradient descent | Good | Slow | High | No |
| **Stochastic** gradient descent | Good (with annealing) | High | Low | Yes |
| **Mini-batch** gradient descent | Good | Medium | Medium | Yes |

Table: Comparison of trade-offs of gradient descent variants

## Challenges

- Choosing a **learning rate**.
- Defining an **annealing schedule**.
- Updating features to **different extent**.
- **Avoiding suboptimal minima**.

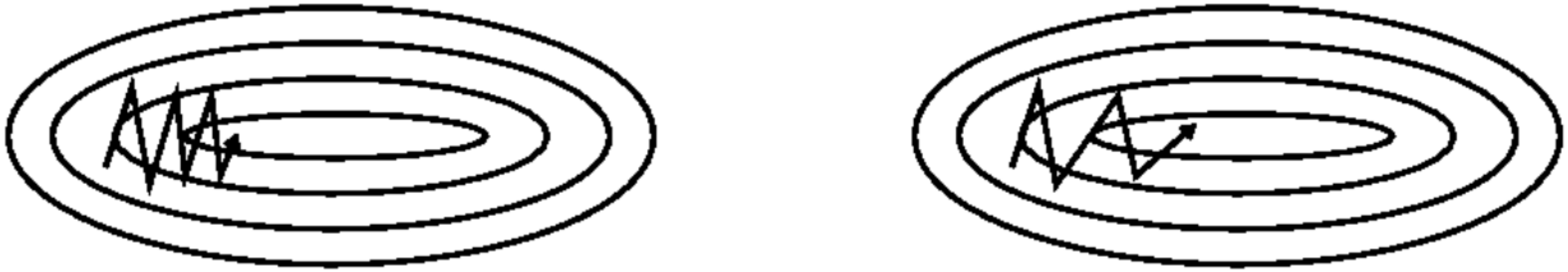
# Gradient descent optimization algorithms

1. Momentum
2. Nesterov accelerated gradient
3. Adagrad
4. Adadelta
5. RMSprop
6. Adam
7. Adam extensions

# Momentum

- SGD has trouble navigating **ravines**.
- Momentum [Qian, 1999] helps SGD **accelerate**.
- Adds a fraction $\gamma$ of the update vector of the past step $v_{t-1}$ to current update vector $v_t$. Momentum term $\gamma$ is usually set to 0.9.

$$\begin{aligned} v_t &= \gamma v_{t-1} + \eta \nabla_\theta J(\theta) \\ \theta &= \theta - v_t \end{aligned} \tag{1}$$



(a) SGD without momentum

(b) SGD with momentum

Figure: Source: Genevieve B. Orr

- **Reduces updates** for dimensions whose gradients **change directions**.
- **Increases updates** for dimensions whose gradients **point in the same directions**.

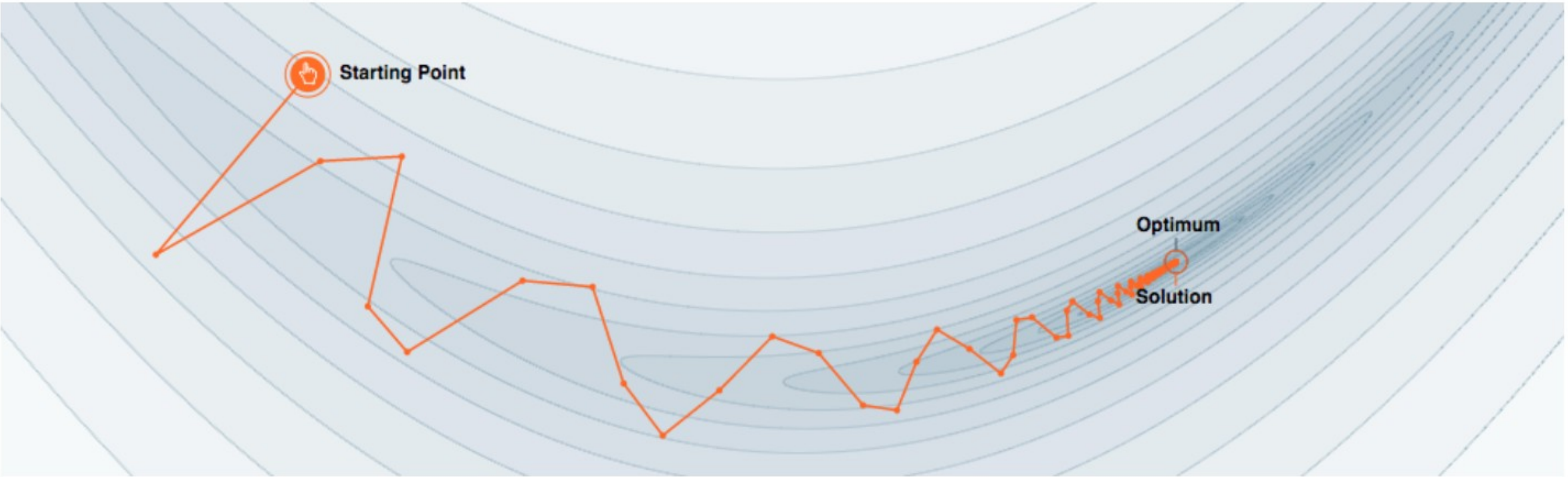
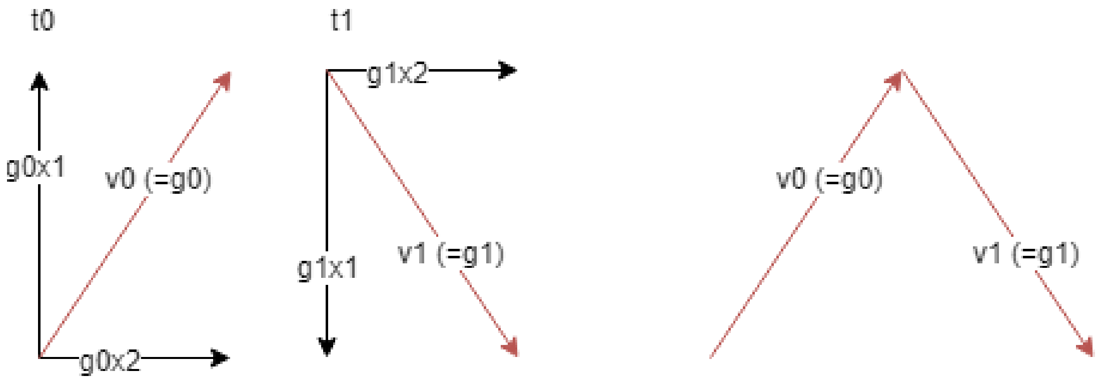
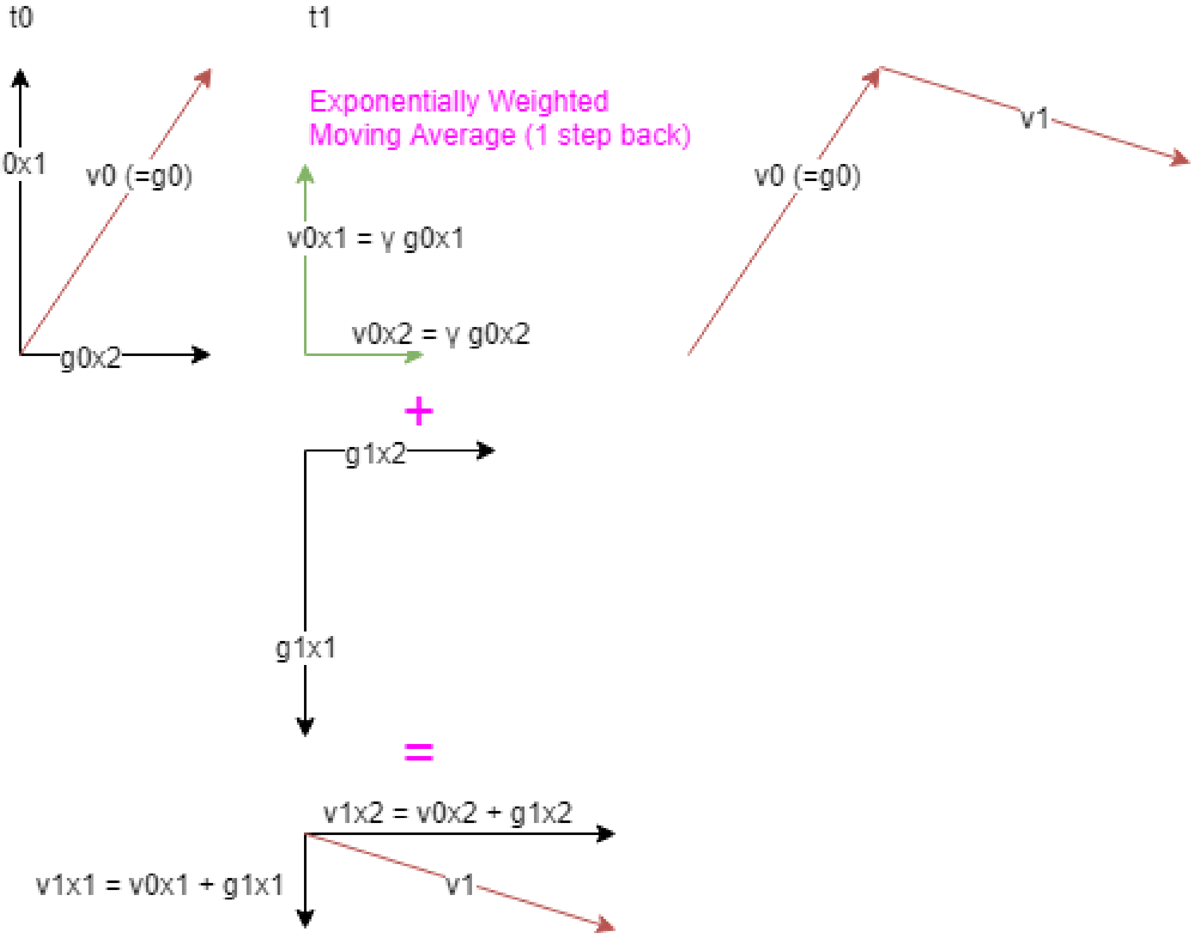


Figure: Optimization with momentum (Source: distill.pub)



Vanilla SGD



SGD with momentum

# Nesterov accelerated gradient

- Momentum **blindly accelerates** down slopes: First computes gradient, then makes a big jump.
- Nesterov accelerated gradient (NAG) [Nesterov, 1983] first makes a big jump in the direction of the previous accumulated gradient $\theta - \gamma v_{t-1}$. Then measures where it ends up and makes a correction, resulting in the complete update vector.

$$\begin{aligned} v_t &= \gamma\, v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1}) \\ \theta &= \theta - v_t \end{aligned} \tag{2}$$

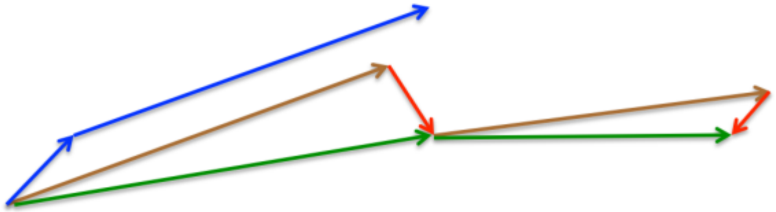


Figure: Nesterov update (Source: G. Hinton's lecture 6c)

# Adagrad

- Previous methods: **Same learning rate** $\eta$ for all parameters $\theta$.
- Adagrad [Duchi et al., 2011] **adapts** the learning rate to the parameters (**large** updates for **infrequent** parameters, **small** updates for **frequent** parameters).
- SGD update: $\theta_{t+1} = \theta_t - \eta \cdot g_t$
  - $g_t = \nabla_{\theta_t} J(\theta_t)$
- Adagrad divides the learning rate by the **square root of the sum of squares of historic gradients**.

|  | θ1 | θ2 | θ3 | θ4 | θ5 | θ6 | θ7 | θ8 |
|---|---|---|---|---|---|---|---|---|
| x1 | 0 | 4 | 0 | 0 | -5 | 0 | -7 | 0 |
| x2 | 0 | 0 | 0 | 0 | 0 | 0 | 10 | 1 |
| x3 | 0 | -5 | 0 | 7 | 0 | 0 | 3 | 0 |
| x4 | 0 | 6 | 0 | 0 | 0 | 0 | -2 | 0 |
| x5 | 0 | 1 | 0 | 0 | 0 | 0 | 5 | 0 |
| x6 | 0 | -8 | 0 | 0 | 4 | 0 | 9 | 0 |

x1-x6 are training samples. When a feature is zero the corresponding parameter does not get updated. Hence, parameters θ7 and θ2 will update frequently, θ5-θ4-θ8 moderately and θ1-θ3-θ6 rarely. Adaptive SGD methods make smaller updates for frequently updating parameters and bigger updates for more rarely updating parameters. For higher hidden layers the relation between θi and inputs is not obvious so to adjust the changing rate we rely on some form of the sum of previous squared gis (gradient values of θi)

# Adagrad

Previously, we performed an update for all parameters $\theta$ at once as every parameter $\theta_i$ used the same learning rate $\eta$. As Adagrad uses a different learning rate for every parameter $\theta_i$ at every time step $t$, we first show Adagrad's per-parameter update, which we then vectorize. For brevity, we set $g_{t,i}$ to be the gradient of the objective function w.r.t. to the parameter $\theta_i$ at time step $t$:

$$g_{t,i} = \nabla_{\theta_t} J(\theta_{t,i})$$

The SGD update for every parameter $\theta_i$ at each time step $t$ then becomes:

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i}$$

In its update rule, Adagrad modifies the general learning rate $\eta$ at each time step $t$ for every parameter $\theta_i$ based on the past gradients that have been computed for $\theta_i$:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$$

$G_t \in \mathbb{R}^{d \times d}$ here is a diagonal matrix where each diagonal element $i, i$ is the sum of the squares of the gradients w.r.t. $\theta_i$ up to time step $t^{11}$, while $\epsilon$ is a smoothing term that avoids division by zero (usually on the order of $1e-8$). Interestingly, without the square root operation, the algorithm performs much worse.

# Adagrad

- Adagrad update:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t \tag{3}$$

  - $G_t \in \mathbb{R}^{d \times d}$: diagonal matrix where each diagonal element $i, i$ is the sum of the squares of the gradients w.r.t. $\theta_i$ up to time step $t$
  - $\epsilon$: smoothing term to avoid division by zero
  - $\odot$: element-wise multiplication

- Pros
  - Well-suited for dealing with **sparse data**.
  - Significantly **improves robustness** of SGD.
  - Lesser need to manually tune learning rate.
- Cons
  - **Accumulates squared gradients** in denominator. Causes the learning rate to **shrink** and become **infinitesimally small**.

# Adadelta

- Adadelta [Zeiler, 2012] restricts the window of accumulated past gradients to a **fixed size**. SGD update:

$$\begin{aligned} \Delta\theta_t &= -\eta \cdot g_t \\ \theta_{t+1} &= \theta_t + \Delta\theta_t \end{aligned} \tag{4}$$

- Defines **running average** of squared gradients $E[g^2]_t$ at time $t$:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2 \tag{5}$$

    - $\gamma$: fraction similarly to momentum term, around 0.9
- Adagrad update:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t \tag{6}$$

- Preliminary Adadelta update:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \tag{7}$$

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \tag{8}$$

- Denominator is just root mean squared (RMS) error of gradient:

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} g_t \tag{9}$$

- Note: **Hypothetical units do not match**.
- Define **running average of squared parameter updates** and RMS:

$$\begin{aligned} E[\Delta\theta^2]_t &= \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma)\Delta\theta_t^2 \\ RMS[\Delta\theta]_t &= \sqrt{E[\Delta\theta^2]_t + \epsilon} \end{aligned} \tag{10}$$

- Approximate with $RMS[\Delta\theta]_{t-1}$, replace $\eta$ for **final Adadelta update**:

$$\begin{aligned} \Delta\theta_t &= -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t \\ \theta_{t+1} &= \theta_t + \Delta\theta_t \end{aligned} \tag{11}$$

# RMSprop

- Developed independently from Adadelta around the same time by Geoff Hinton.
- Also divides learning rate by a **running average of squared gradients**.
- RMSprop update:

$$\begin{aligned} E[g^2]_t &= \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2 \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \end{aligned} \tag{12}$$

  - $\gamma$: decay parameter; typically set to 0.9
  - $\eta$: learning rate; a good default value is 0.001

# Adam

- Adaptive Moment Estimation (Adam) [Kingma and Ba, 2015] also stores **running average of past squared gradients** $v_t$ like Adadelta and RMSprop.
- Like Momentum, stores **running average of past gradients** $m_t$.

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \end{aligned} \tag{13}$$

  - $m_t$: first moment (mean) of gradients
  - $v_t$: second moment (uncentered variance) of gradients
  - $\beta_1, \beta_2$: decay rates

- $m_t$ and $v_t$ are initialized as 0-vectors. For this reason, they are biased towards 0.
- Compute bias-corrected first and second moment estimates:

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t} \end{aligned} \tag{14}$$

- Adam update rule:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t}+\epsilon}\hat{m}_t \tag{15}$$

# Adam extensions

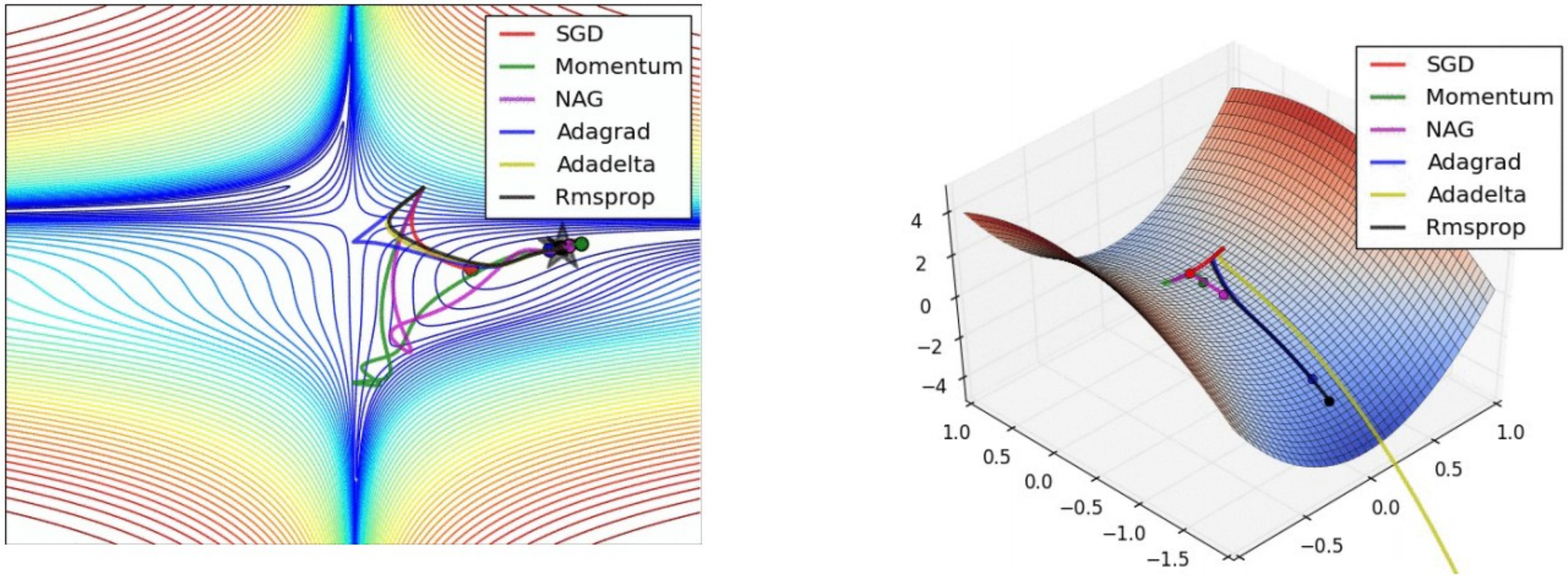
1. AdaMax [Kingma and Ba, 2015]
    - Adam with $\ell_\infty$ norm
2. Nadam [Dozat, 2016]
    - Adam with Nesterov accelerated gradient

# Update equations

| Method | Update equation |
|---|---|
| SGD | $g_t = \nabla_{\theta_t} J(\theta_t)$<br>$\Delta\theta_t = -\eta \cdot g_t$<br>$\theta_t = \theta_t + \Delta\theta_t$ |
| Momentum | $\Delta\theta_t = -\gamma\ v_{t-1} - \eta g_t$ |
| NAG | $\Delta\theta_t = -\gamma\ v_{t-1} - \eta \nabla_\theta J(\theta - \gamma v_{t-1})$ |
| Adagrad | $\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$ |
| Adadelta | $\Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t$ |
| RMSprop | $\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$ |
| Adam | $\Delta\theta_t = -\frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$ |

Table: Update equations for the gradient descent optimization algorithms.

# Visualization of algorithms



(a) SGD optimization on loss surface contours

(b) SGD optimization on saddle point

Figure: Source and full animations: Alec Radford

https://imgur.com/a/Hqolp

# Which optimizer to choose?

- Adaptive learning rate methods (Adagrad, Adadelta, RMSprop, Adam) are **particularly useful for sparse features**.
- Adagrad, Adadelta, RMSprop, and Adam work well in similar circumstances.
- [Kingma and Ba, 2015] show that bias-correction helps Adam **slightly outperform RMSprop**.

# Parallelizing and distributing SGD

1. Hogwild! [Niu et al., 2011]
   - Parallel SGD updates on CPU
   - Shared memory access **without parameter lock**
   - Only works for **sparse input data**
2. Downpour SGD [Dean et al., 2012]
   - **Multiple replicas** of model on subsets of training data run in parallel
   - Updates sent to parameter server; **updates fraction of model parameters**
3. Delay-tolerant Algorithms for SGD [Mcmahan and Streeter, 2014]
   - Methods also adapt to **update delays**
4. TensorFlow [Abadi et al., 2015]
   - Computation graph is split into a **subgraph for every device**
   - Communication takes place using Send/Receive node pairs
5. Elastic Averaging SGD [Zhang et al., 2015]
   - **Links parameters elastically** to a center variable stored by parameter server

# Additional strategies for optimizing SGD

1. Shuffling and Curriculum Learning [Bengio et al., 2009]
   - Shuffle training data after every epoch to **break biases**
   - Order training examples to **solve progressively harder problems**; infrequently used in practice
2. Batch normalization [Ioffe and Szegedy, 2015]
   - **Re-normalizes every mini-batch** to zero mean, unit variance
   - Must-use for computer vision
3. Early stopping
   - "*Early stopping (is) beautiful free lunch*" (Geoff Hinton)
4. Gradient noise [Neelakantan et al., 2015]
   - Add Gaussian noise to gradient
   - Makes model **more robust to poor initializations**

# Bibliography

Paper: Ruder S. (2016) An overview of gradient descent optimization algorithms. arXiv preprint arXiv:1609.04747
Blog article: An overview of gradient descent optimization algorithms
Notebook: Exploring gradient descent based optimizers.ipynb

Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT press. CHAPTER 8